CrossTalk

Jan / Feb 2014 **The Journal of Defense Software Engineering** Vol. 27 No. 1

LEGACY SYSTEM Software Sustainment

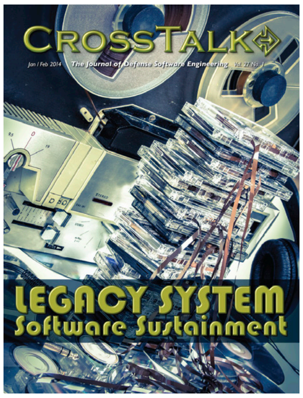
Report Documentation Page	*Form Approved* *OMB No. 0704-0188*

1. REPORT DATE **JAN 2014**	2. REPORT TYPE	3. DATES COVERED **00-00-2014 to 00-00-2014**
4. TITLE AND SUBTITLE **CrossTalk. The Journal of Defense Software Engineering. Volume 27, Number 1. Jan/Feb 2014**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **517 SMXS MXDEA,6022 Fir Ave Bldg 1238,Hill AFB,UT,84056-5820**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT **Approved for public release; distribution unlimited**		
13. SUPPLEMENTARY NOTES		
14. ABSTRACT		
15. SUBJECT TERMS		

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**40**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

TABLE OF CONTENTS


Cover Design by Kent Bingham

Departments

Legacy System Software Sustainment

CrossTalk

NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Publisher Justin T. Hill
Article Coordinator Heather Giacalone
Managing Director David Erickson
Technical Program Lead Thayne M. Hill
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Art Director Kevin Kiernan

Phone 801-777-9828
E-mail Crosstalk.Articles@hill.af.mil
CrossTalk Online www.crosstalkonline.org

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: Office of Cybersecurity and Communications in the National Protection and Programs Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of CrossTalk providing both editorial oversight and technical review of the journal. CrossTalk's mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

Subscriptions: Visit <www.crosstalkonline.org/subscribe> to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the CrossTalk editorial board prior to publication. Please follow the Author Guidelines, available at <www.crosstalkonline.org/submission-guidelines>. CrossTalk does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the authors and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with CrossTalk.

Trademarks and Endorsements: CrossTalk is an authorized publication for members of the DoD. Contents of CrossTalk are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services:
For questions or concerns about crosstalkonline.org web content or functionality contact the CrossTalk webmaster at 801-417-3000 or webmaster@luminpublishing.com.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

CrossTalk is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing <luminpublishing.com>. ISSN 2160-1577 (print); ISSN 2160-1593 (online)

CrossTalk would like to thank 309 SMXG for sponsoring this issue.

The articles in this edition of CrossTalk address the challenges and opportunities associated with Legacy Software sustainment. As DoD budgets continue to shrink, it is becoming increasingly obvious that in many cases the best way to achieve new system capability is through modification of legacy software that is embedded in an existing system. The challenges of sustaining legacy software over many years are diverse and interesting. Since the DoD is not likely to field as many new systems in the future due to reduced budgets, the pressure to keep existing legacy systems operational will probably continue to increase over time.

Numerous core elements of software sustained in the DoD were originally written 20 to 30 years ago. Even though many of the software systems in sustainment today were written two or three decades ago they still struggle with the same challenges of some new developments. Some of the main challenges faced by developers and project managers are related to improving Quality, Predictability, and Reducing Cost and Rework of software. One area that is changing rapidly is Information Assurance (IA) or system security. The changes associated with IA and system security can be significant obstacles to efficient production but are necessary in order to protect systems from intrusion, data loss, and malicious software.

In this edition of CrossTalk there are two articles related to security of systems, Identifying Trustworthiness Deficit in Legacy Systems Using the NFR Approach, and The Transformation of Software Engineering Security. There is one article specifically focused on reducing rework by catching defects earlier in the software lifecycle, that article is: Using Combinatorial Testing to Reduce Software Rework. Additionally there are three articles devoted to various aspects of improving software sustainment of legacy systems and they are: Software Sustainment – Now and Future, Modeling Software Sustainment, and Addressing Software Sustainment Challenges for the DoD.

I hope that this issue of CrossTalk is helpful to many of you because as long as it continues to be helpful to the software industry, it is worth sponsoring this format in order to share good ideas and relevant information.

Karl G. Rogers
Director
309th Software Maintenance Group

Identifying Trustworthiness Deficit in Legacy Systems Using the NFR Approach

Nary Subramanian, University of Texas at Tyler
Steven Drager, Air Force Research Laboratory
William McKeever, Air Force Research Laboratory

Abstract. Trustworthiness is an important emerging requirement for software systems deployed by the U. S. Air Force. Trustworthiness, briefly stated, is the ability of a software system to be safe, secure, and reliable under a normal operating environment. However, most software systems have not been developed with trustworthiness in mind. Therefore, how do we systematically identify deficit in trustworthiness in existing systems so that they may be re-engineered with trustworthiness as a priority? The Non-Functional Requirements (NFR) Approach provides a framework for identifying gaps in trustworthiness in existing systems and recommending mechanisms to overcome this "shortfall" in re-engineered systems. In this project we applied the NFR Approach, as a case study to the middleware system called Phoenix used by the Air Force and determined an 89% shortfall in trustworthiness. The advantages of identifying this deficit include determination of trustworthiness in current systems, exploring environments in which current systems may be (re)used, and prioritizing trustworthiness requirements when these legacy systems are re-engineered.

Introduction

Trustworthiness is an important emerging requirement for software systems including those deployed by the U. S. Air Force: the National Software Strategy Report [1] has concluded that trustworthiness in software will become the most important goal by the year 2015. Trustworthiness, briefly stated, is the ability of a software system to be safe, secure, and reliable under normal operating environments [2]. However, several legacy systems in operation were not designed with trustworthiness in mind—therefore, these systems can be used in a trustworthy environment in one of two ways: employing wrappers that will improve trustworthiness of the system or re-engineering the system to be trustworthy. The second option is a long-term solution but will be expensive in terms of effort and cost required to re-engineer the several dozens of systems being currently used by the Air Force. The first option, namely, the addition of wrappers may be a more cost-effective option for many systems. However, how do we systematically identify deficit in trustworthiness in existing systems so that solutions may be developed? This is especially important when trustworthiness has different connotations for developers, users, and maintainers.

Trustworthiness has been defined differently by different sources, based on their approach to determine trust in a system. For example, in [3] trustworthiness is defined as the degree of confidence that exists that the system meets its requirements, while in [4] defines trustworthiness as a level of confidence of using software engineering techniques to reduce failure rates, enhance testing, reviews and inspections. A discussion of software trustworthiness among stakeholders often invokes numerous non-functional attributes like reliability, safety, usability, portability, or maintainability, which together ensure non-interference with the normal operation of the system.

The NFR Approach [5, 6], where NFR stands for Non-Functional Requirements, provides a framework for systematically analyzing NFRs such as trustworthiness and decomposing it further to capture other NFRs like reliability, safety, portability, etc. The NFR Approach provides the ability to accommodate alternate definitions of trustworthiness as well as provides a rationalization process that allows one to evaluate the extent to which trustworthiness is achieved by a system. More importantly, the NFR Approach helps to identify gaps in trustworthiness requirements. By understanding the extent of "shortfall" of trustworthiness, one is better prepared to identify solutions necessary to make that system trustworthy for a specified time-scale.

In this paper we apply the NFR Approach to a selected software system and identify the trustworthiness deficit in the system. For this purpose we first obtain the definition of trustworthiness for this system from its stakeholders and convert the definitions into a Softgoal Interdependency Graph, an artifact used by the NFR Approach for reasoning about NFRs, which are treated as softgoals in the system. Then the designs for the selected software system are evaluated against trustworthiness definitions using the propagation rules of the NFR Approach. This evaluation will identify deficit in trustworthiness and will permit analysis on how this deficit needs to be overcome. This analysis will help identify adaptations that are needed to make the selected software system function in a trustworthy environment. These adaptations can be stated in terms of design modifications and/or implementation mechanisms (for example, wrappers) that will help the system be used for a specific time-period in a trustworthy environment.

This problem considered is explained by Figure 1: legacy system fulfills primarily its requirements and, mostly by accident, some trustworthy requirements that represent the existing trust in the legacy system. The trustworthy system includes the requirements for trustworthiness that represent the total expected trust as well as the re-engineering requirements for the legacy system. The difference between the total expected trust and the existing trust is the trustworthiness deficit in the legacy system.

The legacy system we used as a case study is the Phoenix middleware system used by the Air Force - we identified the trustworthiness deficit in Phoenix by using the NFR Approach and developed a process for applying this approach to other software systems. Our study identified an 89% shortfall in trustworthiness in the existing Phoenix system.

This paper was presented at the Software Technology Conference held in Salt Lake City, Utah, in April 2013 [7] and was well received by the audience.

Background

The existing approaches to trustworthy analysis split into two categories: product-based and process-based. Product-based techniques [9] identify factors that impact trustworthiness and attempt to satisfy these factors in the product. Process-based techniques, like Trusted Software Methodology [3] and Trustworthy Process Management Framework [10] approach the problem with the belief that trustworthy processes will result in trustworthy products. However, the NFR Approach considers trustworthiness as a non-functional requirement for the product

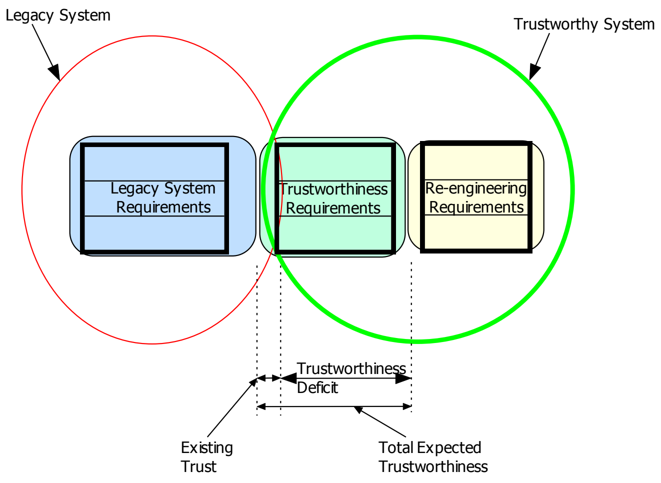

Legacy System
Trustworthy System
Legacy System Requirements
Trustworthiness Requirements
Re-engineering Requirements
Trustworthiness Deficit
Existing Trust
Total Expected Trustworthiness

Figure 1. Context of the Trustworthiness Deficit Problem

being developed; being an NFR its constituents may interact synergistically or conflictingly and NFR Approach is fully geared to analyze these tradeoffs.

The NFR Approach

The NFR Approach is a goal-oriented approach that can be applied to determine the extent to which objectives are achieved by a process or product [5, 6]. NFRs represent properties of a system such as reliability, maintainability, and flexibility, and could equally well represent functional objectives and constraints for a system (NFRs need to be contrasted with functional requirements—the latter state what the software system should do while the former states requirements that are usually observed as a characteristic of the system). In this paper we applied the NFR Approach to design a trustworthy software system by evaluating whether a specific design element satisfied trustworthy requirements for the system. The NFR Approach uses a well-defined ontology for this purpose that includes NFR softgoals, operationalizing softgoals, claim softgoals, contributions, and propagation rules; each of these elements is described briefly below (details may be seen in [5]). Furthermore, the NFR Approach uses the concept of satisficing, a term borrowed from economics, which indicates satisfaction within limits instead of absolute satisfaction, since absolute satisfaction of NFRs is usually difficult.

NFR softgoals represent NFRs and their decompositions. Elements that have physical equivalents (process or product elements) are represented by operationalizing softgoals and their decompositions. Each softgoal is named using the convention (Type [Topic1, Topic2, …]) where Type is the name of the softgoal and Topic (could be zero or more) is the context where the softgoal is used; Topic is optional for a softgoal; for a claim softgoal, which is a softgoal capturing a design decision, the name may be the justification itself.

Following decompositions of either the NFR softgoals or the operationalizing softgoals are possible:

1. AND decomposition is used when each child softgoal of the decomposition has to be satisficed for the parent soft goal to be satisficed but the denial of even one child soft goal is sufficient to deny the parent,
2. OR decomposition is used when satisficing of even one child satisfices the parent but all children need to be denied for the parent to be denied, and
3. EQUAL decomposition has only one child for a parent and propagates the satisficing or the denial of the child to the parent.

Contributions (MAKE, HELP, HURT, and BREAK) are made by operationalizing softgoals to the NFR softgoals and by claim softgoals to other contributions. Reasons for contributions are captured by claim softgoals, and claim softgoals may form a chain of evidence where one claim satisfices another which in turn satisfices another and so on. Each of the four types of contributions has a specific semantic significance: MAKE contribution refers to a strongly positive degree of satisficing of the objectives (represented by NFR softgoals) by artifacts (represented by operationalizing softgoals) under consideration[1], HELP contribution refers to a positive degree of satisficing, HURT contribution refers to a negative degree of satisficing, and BREAK contribution refers to a strongly negative degree of satisficing.

Due to these contributions, some of the softgoals acquire labels that capture the extent to which a softgoal is satisficed: satisficed, weakly satisficed, weakly denied (or weakly not satisficed), denied (or not satisficed), or unknown (indicated by an absence of any label attribute). Moreover, high priority softgoals, decompositions, and contributions may be indicated using the criticality symbol. The graph that captures the softgoals, their decompositions, and the contributions is called the Softgoal Interdependency Graph (SIG). The partial ontology of the NFR Approach is shown in Figure 2. The notations used to indicate the satisficing extent of softgoals are shown in Figure 3.

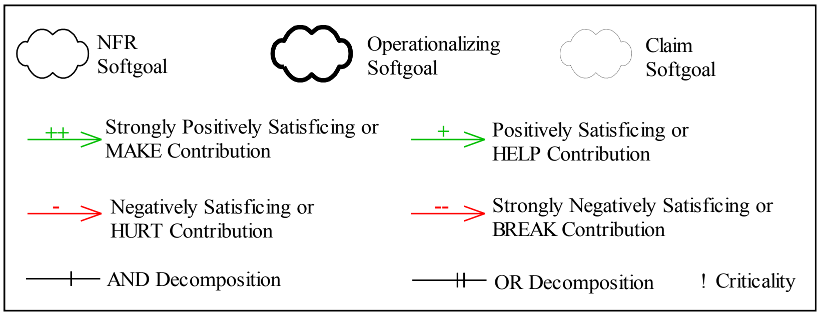

NFR Softgoal
Operationalizing Softgoal
Claim Softgoal
++
Strongly Positively Satisficing or MAKE Contribution
+
Positively Satisficing or HELP Contribution
-
Negatively Satisficing or HURT Contribution
--
Strongly Negatively Satisficing or BREAK Contribution
AND Decomposition
OR Decomposition
! Criticality

Figure 2. Partial Ontology of the NFR Approach

As shown in Figure 2, normal cloud shaped figures represent NFR softgoals, dark-bordered cloud shaped figures represent operationalizing softgoals, and dashed-bordered cloud shaped figures represent claim softgoals. A green arrow annotated with two plus symbols indicates a MAKE contribution, a green arrow annotated with one plus symbol indicates a HELP contribution, a red arrow annotated with a minus symbol indicates a HURT contribution, while a red arrow annotated with two minus

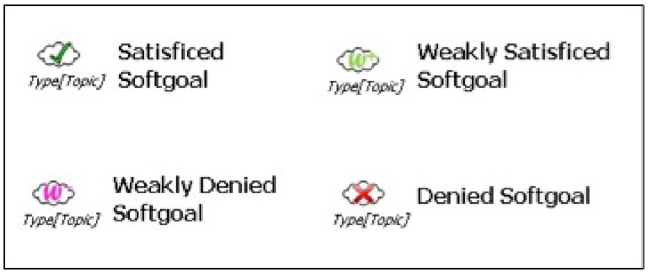


Figure 3. NFR Approach Notations for Softgoal Satisficing

symbols indicates a BREAK contribution. A line with a single cross-line represents AND-decomposition while a line with two cross-lines represents OR-decomposition. Critical elements (softgoals, decomposition, and contributions) are indicated by "!" marks. As shown in Figure 3, a softgoal with a green check mark represents a satisficed softgoal (or a softgoal with a satisficed label), a softgoal with a green W+ annotation represents a weakly satisficed softgoal, a softgoal with a pink W- annotation represents a weakly denied softgoal, and a softgoal annotated with a red X represents a denied softgoal.

Propagation rules propagate labels from child softgoal to the parent across decompositions, from operationalizing softgoals to NFR softgoals across contributions, and from claim softgoals to contributions; propagation rules aid in the rationalization process of the NFR Approach. In a SIG represented graphically, the NFR softgoals and their decompositions are shown at the top of the figure, the operationalizing softgoals and their decompositions are shown in the bottom of the figure, while the contributions between the operationalizing softgoals and the NFR softgoals are shown in the middle. Therefore, contributions are usually received by the leaf NFR softgoals that are at the bottom of the NFR softgoal decomposition hierarchy. While detailed propagation rules may be seen in [5], a simplified list is given below:

R1. A satisficed label is propagated as satisficed by a MAKE contribution, as weakly satisficed by a HELP contribution, as weakly denied by a HURT contribution, and as denied by a BREAK contribution.

R2. A denied label is propagated as denied by a MAKE contribution, as weakly denied by a HELP contribution, as weakly satisficed by a HURT contribution, and as satisficed by a BREAK contribution.

R3. If most of the contributions propagated to a leaf NFR softgoal are satisficed then that NFR softgoal is considered satisficed.

R4. If most of the contributions propagated to a leaf NFR softgoal are denied then that NFR softgoal is considered denied.

R5. In the case of priority softgoals, or when there is a tie between positive and negative contributions, the system architect or the developer can take the decision based on or a variation of R3 and R4

R6. In the case of an AND-decomposition, if all the child softgoals are satisficed then the parent NFR softgoal is satisficed; else the parent softgoal is denied.

R7. In the case of an OR-decomposition, if at least one child softgoal is satisficed then the parent NFR softgoal is satisficed; else the parent softgoal is denied.

R8. In the case of EQUAL-decomposition (only one child) the parent is satisficed if the child is satisficed; and the parent is denied if the child is denied.

Upon applying these propagation rules, if the root (or top-level) NFR softgoals are satisficed then the goals for the domain of interest have been met to a large extent. In this paper the root NFR softgoals will be related to trustworthiness and therefore the SIG will help determine the extent to which a particular design is trustworthy.

The NFR Approach requires the following interleaving tasks, which are iterative

1. Develop NFR goals and their decompositions: in this task the trustworthiness softgoal is decomposed into its constituent NFR softgoals; this decomposition captures the trustworthiness requirements for a system as viewed by a particular group of stakeholders. These decompositions may be developed from scratch or may be extensions of existing decompositions.

2. Develop operationalizing goals and their decompositions: in this task we develop operationalizing softgoals and their decompositions. In this paper operationalizing softgoals correspond to architectural design models[2]. Each individual model may form its own operationalizing softgoal decomposition hierarchy. These models may be developed from scratch or may use existing catalogs as a starting point.

3. Develop goal tradeoffs and rationale: in this task we determine contributions between operationalizing softgoals (task 2) and the NFR softgoals (task 1) and the rationale for the contributions are captured by claim softgoals; synergies and conflicts between different NFR softgoals are captured by the contributions, and tradeoffs (manifested by changes to contributions) that take place are captured by corresponding changes to rationale. This historical record keeping also helps backtracking, if required.

4. Develop goal criticalities: in this task we assign priorities to softgoals—some softgoals (NFR softgoals, operationalizing softgoals, and claim softgoals) may be more important for the stakeholders involved and they are indicated as critical softgoals. Criticalities may also be assigned to decompositions and contributions.

5. Evaluation and analysis: in this task the propagation rules of the NFR Approach are applied to determine whether the design models satisfy the requirements (represented by NFR softgoal decomposition hierarchy) and to what extent – that is, strongly positive, positive, negative, or strongly negative; if positively satisficed then those design models satisfy the requirements and if negatively satisficed then there is scope for improvement.

Example Application of the Steps of the NFR Approach

An example SIG is shown in Figure 4 and we describe how the five steps of the NFR Approach are applied to this SIG. Step 1 involves decomposition of NFR goals for the problem of interest. The upper part of the SIG of Figure 4 captures this decomposition for the NFR trustworthiness for the Phoenix system, which is represented by the root NFR softgoal Trustworthiness [Phoenix]. Based on the definition of trustworthiness for the Phoenix system, we decomposed this NFR softgoal into Dependability [Phoenix], Reliability [Phoenix], Trustworthiness [Phoenix, Software], and Security [Phoenix], which represent, respectively, the requirements that Phoenix must be dependable, Phoenix must be reliable, software for Phoenix must be trustworthy, and Phoenix should be secure. This decomposition is an AND-decomposition, which means all child softgoals must be satisficed for the parent to be satisficed. The NFR softgoal is further AND-decomposed into NFR softgoals Security [Messages] and Timeliness [Messages], which represent, respectively, the requirements that messages be secure and timely. This completes the first step.

In the second step we decompose the design of the Phoenix system. This is shown by operationalizing softgoals at the bottom of Figure 4. We considered two views of the design: Component and Connector Logical View (represented by the operationalizing softgoal C&C View [Logical]) and Detailed Module View of the Submission Service (represented by Module View [Detailed, Submission Service]). The operationalizing softgoal C&C View [Logical] is AND-decomposed into three component softgoals representing Repository Service, Authorization Service, and Channels. The operationalizing softgoal Module View [Detailed, Submission Service] is AND-decomposed into its component softgoals Information Validator, Input Channel Manager, Policy Manager, and Forwarder. This completes the second step.

In the third step of the NFR Approach we determine the contributions between the operationalizing softgoals and the NFR softgoals; these contributions are determined by the domain characteristics. The operationalizing softgoal Repository Service has a MAKE contribution to the NFR softgoal Reliability [Phoenix] and the justification for this contribution is captured by the claim softgoal, "C2 user: repository service provides store-and-forward capability that improves reliability" (here C2 user is one type of system user); this justification gives the rationale for the MAKE contribution. The other three contributions in Figure 4 are BREAK contributions: one between Authorization Service and Security [Messages] with claim softgoal "Limited Authorization", between Channels and Security [Messages] with claim softgoal "Channels do not encrypt messages", and between Information Validator and Security [Messages] with claim softgoal "No authentication or authorization performed". This completes step 3.

In step 4 we can define priorities for NFR softgoals, operationalizing softgoals, claim softgoals, decompositions, and contributions. These priorities depend on the domain requirements. However, for our discussion here we will assume that all elements of the SIG have the same priority.

In step 5 we apply the propagation rules of the NFR Approach to determine the extent of trustworthiness (which is the root NFR softgoal in the SIG) in the Phoenix system. For this purpose we assume[3], based on our current knowledge of the system, all claim softgoals are satisficed. Since all claim softgoals have MAKE contributions, all parent contributions (discussed as part of step 3 above) are satisficed—that is they remain unmodified by the claims. Next we assume, again based on the current knowledge of the domain, that the relevant operationalizing softgoals are satisficed, that is, Repository Service, Authorization Service, Channels, and Information Validator are all satisficed. By propagation rule R6, since all child softgoals of the operationalizing softgoal C&C View [Logical] (the children are Repository Service, Authorization Service, and Channels) are satisficed, the parent C&C View [Logical] is also satisficed. Then by propagation rule R1, four things happen: the satisficed label of Repository Service is propagated as satisficed label to the NFR softgoal Reliability [Phoenix] via the MAKE contribution between them, the satisficed label of Authorization Service is propagated as denied label to Security [Messages] via the BREAK contribution between them, the satisficed label of Channels is propagated as denied label to Security [Messages] via the BREAK contribution between them, and the satisficed label of Information Validator is propagated as denied label to Security [Messages] via the BREAK contribution between them.

Therefore, by propagation rule R4, the NFR softgoal Security [Messages] is denied since only denied labels are propagated to it. Therefore, by R6 the parent NFR softgoal Trustworthiness [Phoenix, Software] is denied since one of its children is denied. By another application of the propagation rule R6 we observe that the topmost NFR softgoal Trustworthiness [Phoenix] is also denied since one of its children is denied—this means that the current design of the Phoenix system is not trustworthy. More importantly, we know why it is untrustworthy since we have the chain of evidence in the SIG: all denied softgoals, decompositions (if any), and contributions indicate the causes for untrustworthiness. Further details of this evaluation may be seen in [11]. Another point to note is that the SIG of Figure 4 was drawn by the StarUML tool with the Softgoal Profile module plugin [8] – this tool automatically applies the propagation rules for a given SIG.

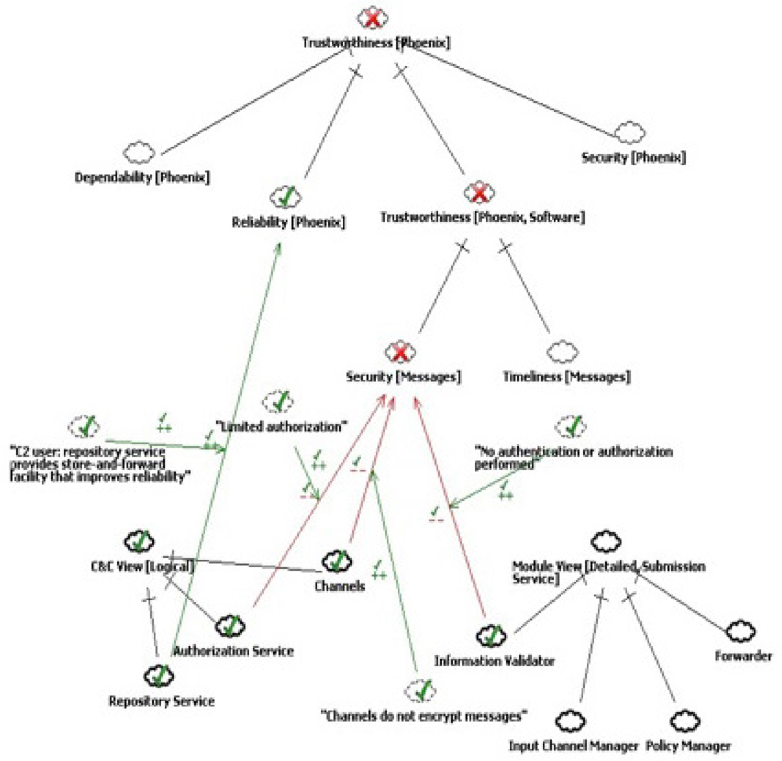

Trustworthiness [Phoenix]
Dependability [Phoenix]
Security [Phoenix]
Reliability [Phoenix]
Trustworthiness [Phoenix, Software]
Security [Messages]
Timeliness [Messages]
"Limited authorization"
"C2 user: repository service provides store-and-forward facility that improves reliability"
"No authentication or authorization performed"
C&C View [Logical]
Channels
Module View [Detailed, Submission Service]
Authorization Service
Information Validator
Forwarder
Repository Service
"Channels do not encrypt messages"
Input Channel Manager
Policy Manager

Figure 4. SIG for Evaluating the Architecture of the Phoenix System for Trustworthiness

Trustworthiness Deficit Identification Using the NFR Approach

In order to identify trustworthiness deficit we need only compare the NFR softgoal decompositions for the untrustworthy and trustworthy system. The actual NFR softgoal decomposition for the Phoenix system is shown in Figure 5. The Phoenix system has interoperable protocols, is reliable has high performance architecture, and is extensible – these are represented, respectively, by the NFR softgoals Interoperability [Protocols], Reliability [Phoenix], Performance [Architecture], and Extensible [Phoenix]. The NFR softgoal Extensible [Phoenix] is AND-decomposed into three child NFR softgoals: Scalability [Architecture], Customizability [Phoenix], and Flexibility [Services], which represent, respectively, scalability of architecture, customizability

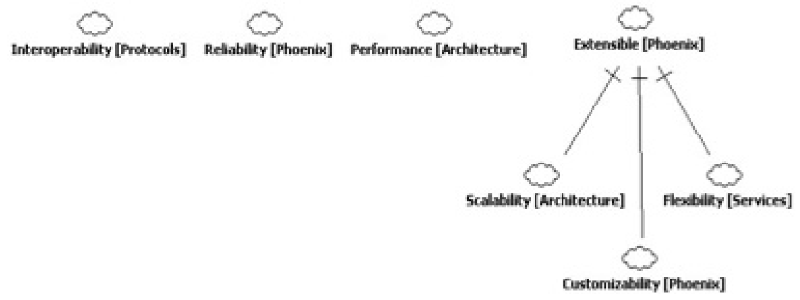

Interoperability [Protocols]
Reliability [Phoenix]
Performance [Architecture]
Extensible [Phoenix]
Scalability [Architecture]
Flexibility [Services]
Customizability [Phoenix]

Figure 5. SIG for Non-Functional Requirements for the Legacy Phoenix System

of Phoenix, and flexibility of services. If we draw a SIG similar to Figure 4, we will find that all of these NFR softgoals are satisficed by the current design of the Phoenix system.

We also obtained the trustworthy requirements for the Phoenix system from the stakeholders—these requirements, from the point of view of one stakeholder, are captured by the SIG of Figure 6. As may be observed, this SIG is the same NFR softgoal decomposition as shown in the upper part of Figure 4, and as we know from the discussion in Section 3, the current design of the Phoenix system does not satisfice these NFR softgoals and is therefore untrustworthy as far as this stakeholder is concerned. In this section we will determine the extent of untrustworthiness by developing the Deficit Equation based on the NFR Approach.

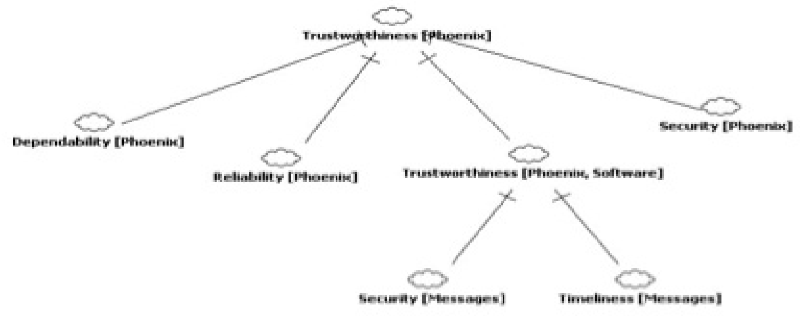


Figure 6. SIG with Trustworthiness Requirements for the Phoenix System from a Stakeholder

These two SIGs (of Figures 5 and 6) may align themselves in three different ways:

1. **Case A: No commonality between the SIGs because they are totally different**
2. **Case B: Some overlap between the SIGs**
3. **Case C: Complete overlap between the SIGs.**

The implications of each of the three possibilities are now discussed. When there is no commonality between the two sets of SIGs as in Case A, this means that the legacy system does not satisfy any trustworthy requirements at all, and has a very high trustworthiness deficit. In Case B, when there is some overlap between the SIGs this means that the legacy system already satisfies some of the trustworthiness requirements, that is, the legacy system is trustworthy to some extent already. Therefore, the trustworthiness deficit in this case is medium, certainly lesser than in Case A. Finally in Case C, when there is a complete overlap between the SIGs, the legacy system already satisfies all trustworthy requirements and the trustworthiness deficit does not exist.

Figure 7 shows the two sets of SIGs for Case A, Figure 8 shows the situation with the SIGs for Case B, and Figure 9 shows the juxtaposition of the SIGs for two scenarios of Case C. The SIGs in Figures 7, 8, and 9, are hypothetical SIGs. SIGs may overlap on individual softgoals or softgoal decompositions. In Figure 8, there is an overlap on two softgoals—that is, these softgoals are common to the legacy system requirements as well as to the trustworthiness requirements. In Figure 9, there is an overlap on softgoal decomposition: in Scenario 1, the overlap is at the root of the SIG, while in Scenario 2, the overlap is at the middle of the SIG. Therefore, the extent of overlap helps identify trustworthiness deficit.

We can quantify this trustworthiness deficit using the following steps:

1. **If no goal overlap occurs, deficit is 100%**
2. **If there is goal overlap, deficit is given by the Deficit Equation, where TS stands for trustworthiness SIG:**

$$deficit = \left(1 - \frac{overlapping\ goals + overlapping\ decompositions}{total\ number\ of\ goals\ in\ TS + total\ number\ of\ decompositions\ in\ TS}\right) \times 100\%$$

Equation 1. Deficit Equation

3. **If there is complete overlap, deficit is 0.**

Therefore, in Figure 7, there are no overlapping goals and no overlapping decompositions while there are four goals and one decomposition in the trustworthy system; therefore, by the deficit equation,

*deficit = (1 − 0/(4+1))*100 = 100%. (for Figure 7 representing case A)*

Therefore, the deficit is 100% in Figure 7. In Figure 8, there are two overlapping softgoals, no overlapping decompositions, four softgoals and one decomposition in the trustworthy system. Therefore,

*deficit = (1 − 2/5)*100 = 60%. (for Figure 8 representing case B)*

In Figure 9, for scenario 1, there are four overlapping softgoals, one overlapping decomposition; therefore,

*deficit = (1-5/5)*100 = 0%. (for Figure 9, scenario 1, representing case C)*

In Figure 9, for scenario 2, the same situation like scenario 1 is obtained and the deficit is again 0%. That the deficit is 0% for both scenarios of Figure 9 should not be surprising since the original system satisfies all trustworthiness requirements.

Likewise, in the SIG of Figure 6, there are seven softgoals and two decompositions in the trustworthiness SIG. Also, comparing the SIGs of Figure 5 and Figure 6, we find that there is only one softgoal in common, namely, Reliability [Phoenix]. Therefore, the trustworthiness deficit is shown in Equation 2.

$$deficit = \left(1 - \frac{1}{9}\right) * 100 = 89\%$$

Equation 2.

Therefore, the trust deficit in the legacy Phoenix system is relatively high. The missing trustworthiness requirements are given in Table 1. As can be seen in Table 1, six requirements come from softgoals and two from softgoal decompositions. These missing requirements will allow us to identify the environments the legacy software system may be safely used in.

Therefore, the process (or checklist) for identifying trustworthiness deficit using the NFR Approach is as follows:

1. Obtain legacy system requirements; create the SIG
2. Obtain trustworthiness requirements; create the SIG
3. Identify extent of overlap between legacy system requirements SIG and trustworthiness SIG
4. Apply the Deficit Equation to evaluate trustworthiness deficit
5. Identify missing trustworthiness requirements – both from softgoals and decompositions in the trustworthiness SIG.

In the first step obtain the requirements for the legacy system either by reverse engineering or from system documentation, then create the SIG with, if needed, stakeholder involvement. Then obtain the trustworthiness requirements for the re-engineered system and create the SIG, again, if needed, with stakeholder involvement. Then identify the extent of overlap between the two SIGs. Apply the Deficit Equation to identify the extent of the deficit. Then identify the missing trustworthiness requirements in the legacy system from softgoals and decompositions in the trustworthiness SIG.

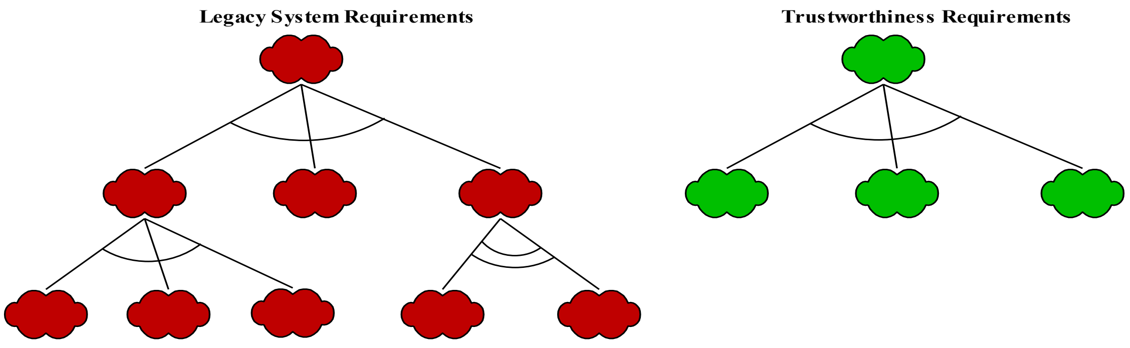


Figure 7. SIGs for Case A: No Commonality, High Deficit

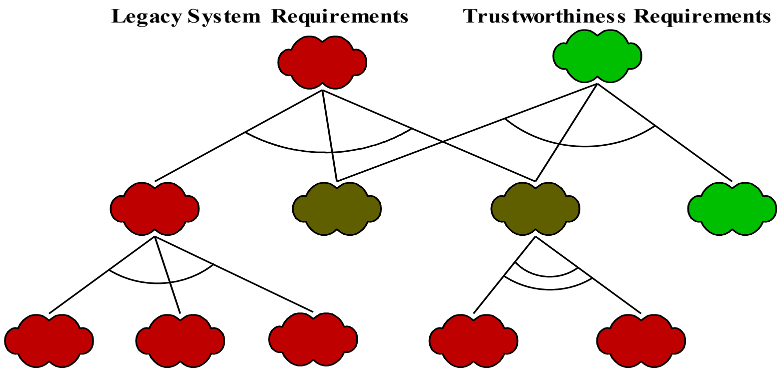


Figure 8. SIGs for Case B: Some Commonality, Medium Deficit

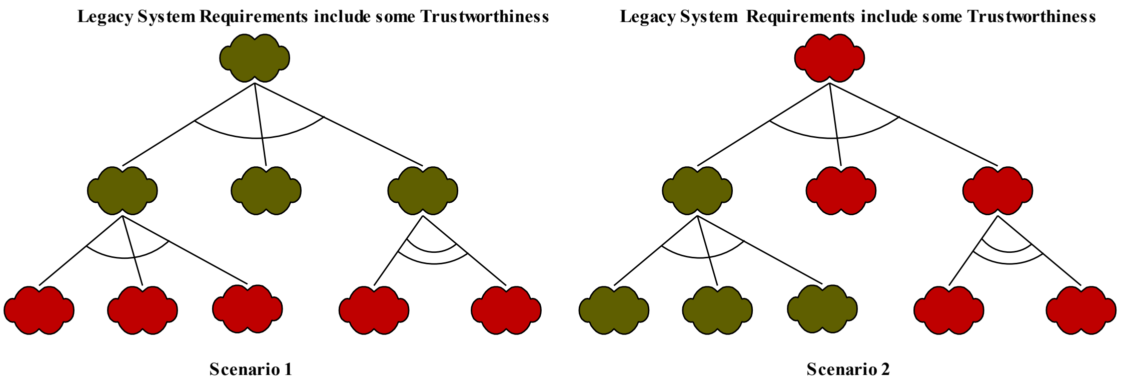

Legacy System Requirements include some Trustworthiness
Legacy System Requirements include some Trustworthiness
Scenario 1
Scenario 2

Figure 9. SIGs for Case C: Complete Overlap, Zero Deficit, Two Scenarios

No.	Missing Trustworthiness Requirements	Source
1	Phoenix system should be trustworthy.	Softgoal: Trustworthiness [Phoenix]
2	Phoenix system should be dependable.	Softgoal: Dependability [Phoenix]
3	Phoenix system should be secure.	Softgoal: Security [Phoenix]
4	Phoenix system software should be trustworthy.	Softgoal: Trustworthiness [Phoenix, Software]
5	Phoenix system should send messages securely.	Softgoal: Security [Messages]
6	Phoenix system should send messages in a timely manner.	Softgoal: Timeliness [Messages]
7	Trustworthy Phoenix system should be dependable, reliable, have trustworthy software, and be secure.	Decomposition: Trustworthiness [Phoenix] is AND-decomposed into Dependability [Phoenix], Reliability [Phoenix], Trustworthiness [Phoenix, Software], and Security [Phoenix].
8	Trustworthy Phoenix system software should send messages securely as well as in a timely manner.	Decomposition: Trustworthiness [Phoenix, Software] is AND-decomposed into Security [Messages] and Timeliness [Messages].

Table 1. Missing Trustworthiness Requirements in the Legacy Phoenix System

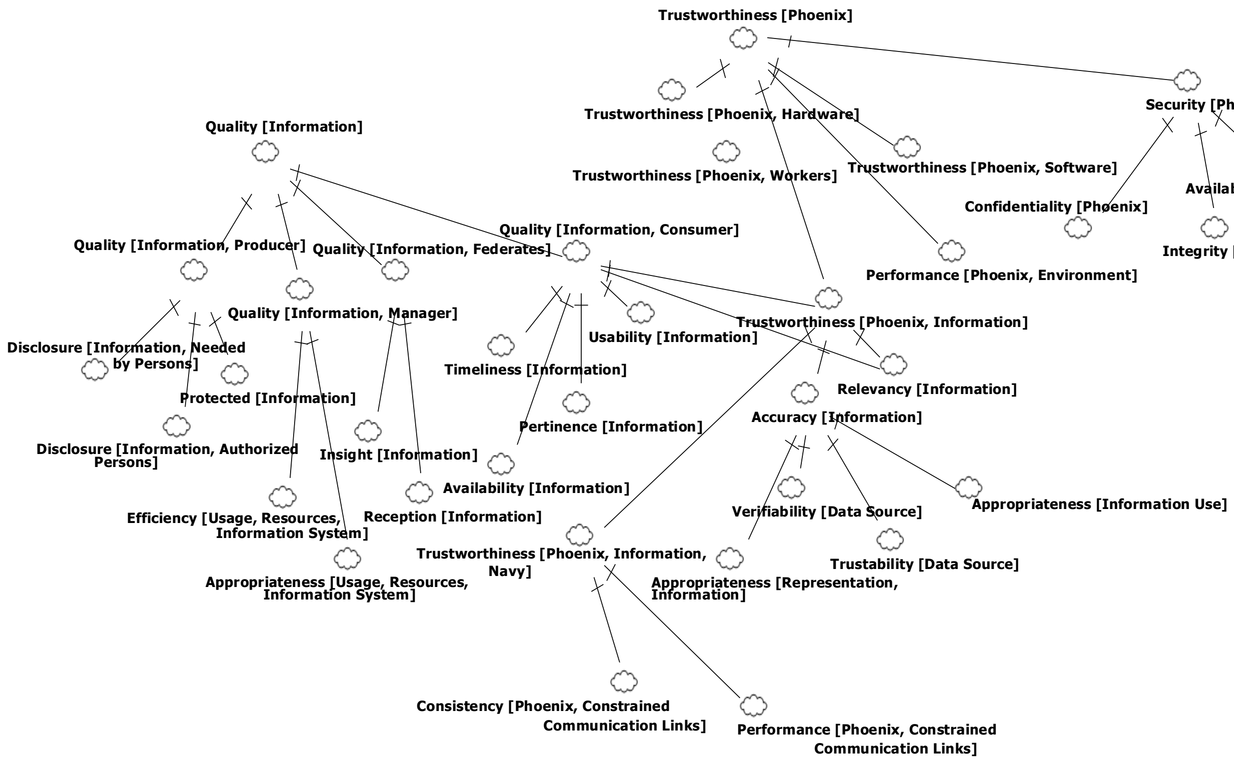

Trustworthiness [Phoenix]
Trustworthiness [Phoenix, Hardware]
Security [Phoenix]
Quality [Information]
Trustworthiness [Phoenix, Workers]
Trustworthiness [Phoenix, Software]
Availability [Phoenix]
Confidentiality [Phoenix]
Integrity [Phoenix]
Quality [Information, Consumer]
Quality [Information, Producer]
Quality [Information, Federates]
Performance [Phoenix, Environment]
Quality [Information, Manager]
Trustworthiness [Phoenix, Information]
Usability [Information]
Disclosure [Information, Needed by Persons]
Timeliness [Information]
Relevancy [Information]
Protected [Information]
Accuracy [Information]
Pertinence [Information]
Disclosure [Information, Authorized Persons]
Insight [Information]
Availability [Information]
Verifiability [Data Source]
Appropriateness [Information Use]
Efficiency [Usage, Resources, Information System]
Reception [Information]
Trustability [Data Source]
Trustworthiness [Phoenix, Information, Navy]
Appropriateness [Usage, Resources, Information System]
Appropriateness [Representation, Information]
Consistency [Phoenix, Constrained Communication Links]
Performance [Phoenix, Constrained Communication Links]

Figure 10. Another Definition of Trustworthiness for the Phoenix System

We mentioned earlier that NFR Approach helps us analyze reasons for poor trustworthiness as well. This analysis proceeds from the SIG of Figure 4 where we see that the main reason for poor trustworthiness is the denial of the NFR softgoal Security [Messages]; this is contributed by three design elements (as discussed in Section 3), which are Authorization Service and Channels of the Component and Connector View and the Information Validator in the Submission Service. Therefore, any improvement in securing messages in all of the three design elements will significantly improve trustworthiness of the Phoenix system. Further details of this analysis may be seen in [11].

It should be noted that the definition of trustworthiness shown in Figure 6 is the view of one stakeholder. Another stakeholder gave the definition of trustworthiness shown in Figure 10, which as can be seen is more complicated. However, the checklist given above can be applied to this definition as well and the trustworthiness deficit can be identified. However, we did not find one single set of attributes that defined trustworthiness acceptable to all stakeholders. As such, NFR Approach provides a process for identifying trustworthiness deficit given any definition of trustworthiness.

Conclusion

Trustworthiness is expected to be an important requirement for software systems in the future. However, not all legacy systems were developed with trustworthiness in mind. It will be helpful if we could systematically identify gaps in trustworthiness in a software system so that the suitability of the software system for use in trustworthy environments may be determined. This is also important to understand the environments where the software system may be used or re-used as well as to determine the requirements that need prioritizing when the software system is being re-engineered. We applied the NFR Approach [5, 6] for this trustworthiness deficit identification since the NFR Approach is useful in dealing with non-functional requirements (NFRs) such as trustworthiness. The NFR Approach considers trustworthiness as a goal to be achieved by the software system and identifies the deficit by determining the extent to which the system falls short of the goal.

In order to develop a process by which NFR Approach may be systematically applied to any software system, we applied it, as a case study, to the Phoenix system. The Phoenix system is a middleware system used by the Air Force with about 100,000 lines of code. We first obtained the current requirements (or legacy requirements) satisfied by the Phoenix system. We then obtained the trustworthiness requirements for the Phoenix system from the stakeholders. Then applying the NFR Approach we determined the trustworthiness deficit in the Phoenix system to be 89% - that is, the system is highly untrustworthy. Based on this case study we believe that the process of the NFR Approach can be applied to any software system to identify its trustworthiness deficit.

For the future we plan to extend the deficit equation to include both hardgoals and softgoals—that is, consider both functional and non-functional requirements [12]. We also plan to apply the NFR Approach to larger systems than Phoenix and confirm that the NFR Approach is scalable to larger systems. We also plan to quantitatively assess trustworthiness in a software system [13] so that changes to design may be motivated by quantitative considerations.

Disclaimer:

Approved for Public Release [88ABW-2013-4662] 07Nov13, Distribution unlimited. ❖

ABOUT THE AUTHORS

Nary (Narayanan) Subramanian is currently an Associate Professor of Computer Science at The University of Texas at Tyler, Tyler, Texas. Dr. Subramanian received his Ph.D. in Computer Science from The University of Texas at Dallas. His specialization is software engineering with particular focus on software architectures and requirements engineering. He co-founded the International Workshop on System/Software Architectures (IWSSA) and served as a co-chair for seven years between 2002 and 2011. He established and directed the Center for Petroleum Security Research at UT Tyler. He has over fifteen years' experience in industry in engineering, sales, and management. He is a member of the IEEE. His research interests include software engineering, system engineering, and security engineering.

Department of Computer Science
University of Texas at Tyler
Tyler, Texas, USA
E-mail: nsubramanian@uttyler.edu
Phone: 903-566-7309

Steven Drager is a principal electronics engineer with the Air Force Research Laboratory Advanced Computing and Communications Division leading research in trusted computing, high performance systems and emerging models and technologies for computation. Mr. Drager has over 20 years at AFRL beginning in reliability physics working wafer-level testing for oxide breakdown, hot carrier degradation and electromigration and then on the development and standardization of the analog and mixed-signal extensions to the VHSIC Hardware Description Language (VHDL-AMS). He has spent the last 10 years leading research in high performance embedded computing architectures, quantum computing architectures and algorithms, and software-intensive systems producibility.

Information Directorate
Air Force Research Lab
Rome, New York, USA
E-mail: Steven.Drager@us.af.mi
Phone: 315-330-2735

William McKeever has been with the Air Force Research Laboratory's Information Directorate, since 2003. He is co-lead of the Trusted Software-intensive Systems research which seeks to develop techniques, methodologies and tools to guarantee trust (as measured by correctness, security, reliability, predictability, and survivability)and migrate the analysis from execution (testing and monitoring) to design (correct and formal/security specifications) and development (composition and auto-generation) to meet DoD System needs. Mr. McKeever received a BS in Computer Science from Plattsburgh State University of New York, and a MS in Computer and Information Science from State University of New York Institute of Technology.

Information Directorate
Air Force Research Lab
Rome, New York, USA
E-mail: William.Mckeever.1@us.af.mil
Phone: 315-330-2897

REFERENCES

1. <http://www.cnsoftware.org/NSS2Report>
2. NIST Trustworthy Information Systems program available at <http://www.nist.gov/itl/tis/>.
3. E. Amoroso, et al "A Process-oriented Methodology for assessing and improving software trustworthiness", Proceedings of the 2nd ACM Conference On Computer and Communication Security, 1994, pp. 39-50.
4. D. L. Parnas et al "Evaluation of safety-critical software", Communications of ACM, Volume 33, Issue 6, 1990, pp. 636-648.
5. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, Non-Functional Requirements in Software Engineering, Kluwer Academic Publishers, Boston, 2000.
6. N. Subramanian and L. Chung, "Software Architecture Adaptability - An NFR Approach", Proceedings of the International Workshop on Principles of Software Evolution (IWPSE 2001), ACM Press, Vienna, September, 2001, ACM Press, pp. 52-61.
7. N. Subramanian, S. Drager, and W. McKeever, "Identifying Trustworthiness Deficit in Legacy Systems Using the NFR Approach", Software Technology Conference, Salt Lake City, Utah, April, 2013.
8. <http://staruml.sourceforge.net/en/modules.php accessed on August 10, 2013>.
9. D. Taibi, "Defining a Open Source Software Trustworthiness Model", Proceedings of 3rd International Doctoral Symposium on Emperical Software Engineering, 2008.
10. Y. Yang, Q. Wang, M. Li, "Process Trustworthiness as a capability indicator for measuring and improving software trustworthiness", ICSP '09 Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes, pp 389-401.
11. N. Subramanian, S. Drager, W. McKeever, "Designing Trustworthy Software Systems using the NFR Approach", Chapter Paper, to appear in Emerging Trends in ICT Security, Edited by Babak Akhgar and Hamid Arabnia, Elsevier Publication, 2014.
12. L. Chung et al., "Goal-Oriented Software Architecting", Relating Software Requirements and Architectures, (Eds.) P. Avgeriou et al., Springer, 2011, pp. 91-109.
13. N. Subramanian, S. Drager, and W. McKeever, "Evaluating Trustworthiness Using the NFR Approach" poster presented at the Cyber and Information Challenges Conference in June 2012, organized by Armed Forces Communications and Electronics Association, in Utica, NY.

NOTES

1. When contributions (MAKE, HELP, HURT, or BREAK) are between claim softgoals and other contributions, then the "objectives" are these other contributions and the "artifacts" are the justifications captured by claim softgoals.
2. The NFR Approach supports any level of realization: strategic level, conceptual level, system level, requirements level, architectural design level, detailed design level, code level, and so on; however, in this paper we considered architectural design models.
3. If this assumption changes at any time we update the SIG to reflect the changes and re-evaluate by applying the propagation rules.

Second Generation Product Line Engineering Takes Hold in the DoD

Paul Clements, BigLever Software,
Susan P. Gregg, Lockheed Martin,
Charles Krueger, BigLever Software,
Jeremy Lanman, U.S. Army PEO STRI,
Jorge Rivera, General Dynamics,
Rick Scharadin, Lockheed Martin,
James T. Shepherd, Lockheed Martin,
Andrew J. Winkler, Lockheed Martin

Abstract. Product Line Engineering (PLE) is a well-established engineering discipline that provides an efficient way to build and maintain portfolios of systems that share common features and capabilities. Systems—including DoD systems—built with PLE have, for decades now, demonstrated improvements in development time, cost, quality, and engineering productivity that consistently attain integer-multiple improvements over comparable non-PLE engineering efforts. Until recently there was no unified repeatable approach available; each PLE project went its own way. But now, two high-visibility DoD examples (Navy's AEGIS and Army's Live Training Transformation) are taking advantage of a strong and well-defined automation-centered approach that some are calling Second Generation PLE, and reaping substantial benefits as a result.

Introduction

The DoD is rife with systems that share much in common. For example, over 80 companies, universities, and government organizations are actively developing one or more of some 200 unmanned aerial vehicle designs. They differ from each other in important ways, but they resemble each other in ways that are at least as important. In 2004, the General Accounting Office was able to identify 2,274 separate DoD business systems (but nobody knows the true number) that are different, but also alike. The Joint Strike Fighter is being delivered in three main variants with very different capabilities, but they are all still the F-35. Communication systems, armored vehicles, tactical fixed-wing aircraft, helicopters—the list of large-scale examples of systems that are different yet the same goes on and on.

These examples are—or in many cases should be—product lines. A product line is a set of systems that share common features, and are engineered, developed, and sustained using a common set of shared assets[1]. The systems are built and maintained in a way that respects the variations in capability and function that they each need to provide to their respective users, but also takes maximum advantage of the commonality they share. PLE is the name of the established engineering discipline that far-sighted organizations use to accomplish this. It is an efficient way of building and maintaining portfolios of systems.

This article is about two high-visibility examples in the DoD where far-sighted organizations are achieving that efficiency. The AEGIS command and control systems of Naval surface combatants differ widely, but have so much in common with each other that it is more beneficial to consider them as variants in the same family. The Army's Live Training Transformation comprises a multitude of training systems covering a spectrum from single-soldier weapons trainers to large-scale synthetic force-on-force wargaming systems. Once again, there is benefit being gained by viewing them as a family.

PLE: Feeling Its Way in the First Generation

Systems built under the discipline of PLE have, for decades now, experienced improvements in development time, cost, quality, and engineering productivity that consistently attain integer-multiple improvements over previous engineering efforts. The PLE community, eager to spread the word, has over the years published a swarm of readily available case studies and catalogs of successful PLE-engineered families of systems in industry [14][3][9][12][15]. Many of the improvements reported are jaw-dropping, such as a family of embedded engine controllers that used to take a year to develop and under PLE take less than a week [3], or a family of computer peripherals can be built with 1/4 of the staff, in 1/3 of the time, and with 1/25 the number of bugs as the organization's pre-PLE products [14].

However, each of these successes employed its own unique approach and techniques applied atop the basic concepts in varying degrees and in varying ways. These approaches, which can be characterized as first-generation, were point-case effective but lacked a systematic, repeatable, codified methodology. All made a strong distinction between domain engineering (creation of reusable parts) and its equal counterpart application engineering (creation of specific products from those parts), focused on software code as the most important reusable resource, and used the concept of a feature to compare systems in a domain.

Nevertheless, the benefits were real and attention-grabbing. In addition to the hard numbers, PLE practitioners have consistently reported a wide array of less tangible (but arguably no less important) benefits, including:

- **Ability to perform continuous portfolio-wide insertion of new technology and new functionality at low cost**
- **Uniform look and feel to products and greater interoperability**
- **Higher engineer satisfaction with resulting lower workforce turnover**

This message was not lost on the Pentagon or its contractors, both eager to lower cost and to translate (for example) reduced time to market into reduced time to deployment to support the Warfighter. Some early but notable examples of DoD-oriented product line efforts include:

- **A product line of satellite ground control systems for the National Reconnaissance Office [3]**
- **A product line of weapons test ranges at the Naval Undersea Warfare Center [4]**
- **A product line of helicopter avionics systems for the Army's Technical Applications Program Office [2]**
- **A product line of submarine combat systems for the Navy's Submarine Warfare Federated Tactical System [8]**

These efforts, too, enjoyed the same kind of eye-catching benefits: Millions of dollars saved, delivery times slashed, and increased capability for lower cost.

Meanwhile, PLE as a discipline was evolving. Languages for expressing variation became more uniform and simpler, reflecting only what was needed in practice. Automation to support product derivation from shared assets moved out of the research labs and into real-world application, gaining robustness, simplicity, and usability. PLE adopted a whole-system perspective, a powerful generalization reflecting a move away from the field's software-only roots. The trends have crystallized into an approach some are calling "Second Generation Product Line Engineering" (2GPLE) [7].

PLE: Second Generation Maturation

Building on first-generation efforts, 2GPLE embodies a more well-defined and repeatable process, centered on a strong factory paradigm. Distinguishing characteristics of 2GPLE include:

1. Features express product variation: In the factory paradigm, we need a way to describe what product we are building, so the shared assets (requirements, designs, code, test cases, user manuals, etc.) can be configured appropriately. Rather than adopt a different "language" and mechanism for each type of artifact (for example, compiler directives for code, attributes for requirements, text variables for documents, and so forth), 2GPLE uses a small and consistent set of variation mechanisms [1] for all of the artifacts. Each product is described by giving a list of its features: "A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems" [10]. Features are used to express product differences in all lifecycle phase artifacts. This streamlines the development process and lets all stakeholders speak the same language.

2. Shared assets come from all lifecycle phases, not just the software: Early approaches to PLE certainly encouraged practitioners to include all kinds of artifacts in their collection of shared assets, but the unmistakable emphasis was on software. But in large-scale product lines, automated production of whole and consistent sets of lifecycle artifacts is essential. Managing these artifacts means imbuing them with variation points [1], which are places where an artifact can change to support different products. Variation points reflect the different feature-based product contexts in which the artifacts will be used. In 2GPLE, all supporting assets are considered equally important; software plays the same role as any other, or even (in cases where the products contain no software) no role at all.

3. Industrial-strength automation is employed in the form of a configurator, which is a tool that takes a feature-based description of a product and exercises the variation points in the shared assets to produce an artifact set that supports the named features. Product development thus becomes automated, so that application engineering (so important in first-generation approaches) becomes vanishingly small. (Both product line organizations in this article chose the BigLever Software Gears PLE configurator [11] as the automation engine to power their product line.)

Figure 1 illustrates these three distinguishing aspects of 2GPLE. A feature profile is a description of a product in terms of the feature choices. The configurator (here, Gears) uses the feature profile to configure each shared asset (by exercising its variation points) to produce the set of engineering artifacts specific to that product.

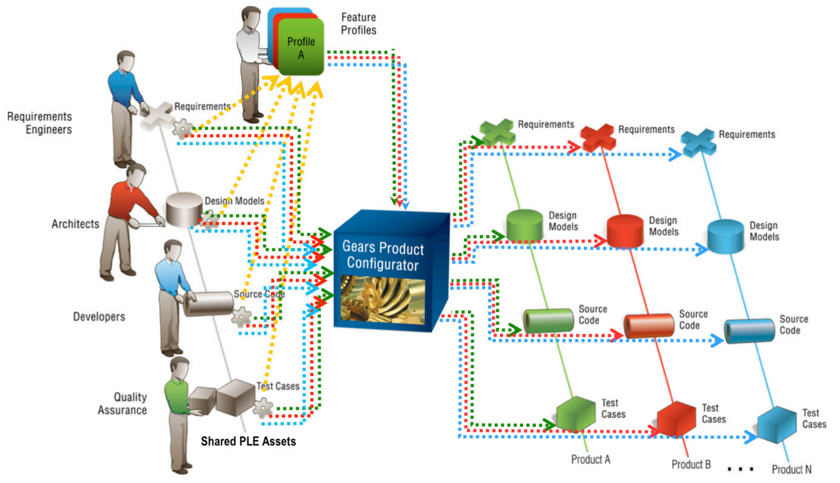


Figure 1. The 2GPLE factory paradigm. The configurator uses a feature profile for a product to exercise variation points (denoted by the gear symbols) in the shared assets, configuring them to support a product with those features.

PLE IS MUCH MORE THAN REUSE

To understand what PLE is, it is important to understand what it is not. A superficial explanation of PLE describes reuse through shared artifact repositories. Yes, there is reuse, and yes, there are repositories, but that is like explaining Project Apollo by starting with powdered orange breakfast beverage. It was there, but was hardly the point.

Many organizations claim, incorrectly, that they are employing PLE when in fact are only practicing reuse and nothing more. And they are practicing a particularly problematic form of reuse called "clone and own."

Figure 4 shows a stylized view of a production shop in which N products are developed and maintained—or, for that matter, acquired. This "shop" could turn out the systems under a PEO's purview, and be run by a single contractor, or a prime with subs, or separately administered programs. In this simplified view, each product comprises a set of artifacts; for example, requirements, design models, source code, and test cases. Each engineer in this shop works primarily on a single product. When a new product is launched, its project copies—clones—the most similar assets it can find, and starts adapting them to meet the new product's needs. Development and acquisition efforts that think reuse is the goal can chalk up impressive metrics to claim success.

But under this kind of reuse, making portfolio-wide changes becomes prohibitively expensive. And portfolio-wide changes are the norm in DoD systems: New hardware, new architectures, new standards, new mission doctrines, new rules of engagement, new systems to interoperate with, new adversaries, and new threats can easily lead to the need to change every system in a family.

To see how clone-and-own reuse can lead to intractable complexity, consider one kind of portfolio-wide change: Defect elimination. Assume that a defect is found in Product B and that the defect is traced to an ambiguous or incorrect requirement in Product B's requirements. The Product B team fixes the error, re-designs as necessary, then fixes the code and test cases before re-deploying Product B. Product B is now healthy again.

But suppose that the defect in Product B's requirements was "inherited" when the Product B team copied the requirements from Product A. Suppose further that the source code for Product N was copied from Product B's (defective) source code, and the test cases for Product N were similarly "borrowed" from Product N's (inadequate) test cases.

To really root out the defect from the entire portfolio, each of the N product teams should really confer with each of the other N-1 product teams. These communication paths are shown in red in Figure 4. This communication obligation imposes an overhead that grows as the square of the number of products. So, in a relatively modest product line of 30 products, almost 900 inter-project communication paths should be activated. This complexity will quickly overwhelm any program office, let alone any engineering staff, and the result is usually exhaustion, a climbing defect rate, out-of-control sustainment cost, and a reluctance or inability to make changes.

This complexity occurs even if reuse levels are as high as possible among the programs; the product line will still collapse under the weight of its "clone and own" reuse strategy. Copy-based reuse gives the copying program a head start, but then loses all of its value as the new program spirals off on its own evolution and sustainment trajectory. Acquisition programs that encourage reuse but not true product line engineering are setting themselves up for sustainment failure.

The automation-centered approach also enables a fourth salient characteristic of 2GPLE: A simplified model for configuration management. The shared assets are configuration-controlled, but the products need not be, since they can be quickly re-generated [11].

A fifth characteristic involves feature languages that facilitate modular and hierarchical product lines developed across organizational boundaries [7]. This allows a system-of-systems family to become a product-line-of-product-lines.

Overall, 2GPLE represents a more clearly formulated methodology that organizations can use directly. It simultaneously generalizes and simplifies concepts from its first-generation roots. Once again, industry and the DoD are paying attention. In addition to 2GPLE projects in industry at large—General Motors, for instance [7]—two multi-billion-dollar high-visibility programs in the Army and the Navy (respectively) are employing 2GPLE to help their Warfighters train and fight, and are seeing substantial benefits in reliability, sustainability, and responsiveness. The two programs are Live Training Transformation and AEGIS.

2GPLE in the Army: Live Training Transformation

In 2010 General Dynamics teamed with BigLever Software (the PLE technology provider) to create the winning proposal for the US Army's Live Training Transformation (LT2) family of training systems. (This contract was the first U.S. Army contract focused specifically on product line engineering as a required part of the solution.)

The United States Army Program Executive Office for Simulation, Training and Instrumentation (PEO STRI) is in the business of training soldiers and growing leaders by providing responsive, interoperable simulation, training, and testing solutions and acquisition. Its training and testing systems portfolio includes live, virtual, and constructive training packaged in embedded and interoperable products that are fielded and used throughout the world.

LT2 has long been a true software product line, in the sense defined in [3], using first-generation approaches. In 2010 the program made the transition to 2GPLE. LT2 shared assets include the open architectures, common software components, standards, processes, policies, governance, documentation, and more, all leading to a common approach and frameworks for developing live training systems. Examples of the many types of training systems in the LT2 family include Military Operations on Urban Terrain (MOUT), Maneuver Combat Training Center (MCTC), instrumented live-fire range training, and various Joint (that is, inter-Service) training systems.

The commonality behind LT2 facilitates the rapid development of new products but also ensures that products across the LT2 product line can communicate and interoperate with each other. This is important because large training exercises need to employ different kinds of training systems working together. The LT2 product line makes use of plug and play components and applications that are common between products, and permits changes, upgrades and fixes developed for one product to be applied to others. This concept provides the inherent logistics support benefits that derive from commonality, standardization, and interoperability including the reduction of total lifecycle costs [13].

The LT2 migration to 2GPLE is proving easier than expected. First, a product line culture and high reuse were already in place with the first generation product line. Second, 2GPLE approaches are easier to adopt because they enable non-disruptive and incremental steps to be taken rather than a large "big bang" start-over event. LT2 stakeholders have already enjoyed substantial benefits from LT2's first-generation approach and are therefore more willing to move to 2GPLE.

Maximizing asset sharing has proven to reduce fielding time and minimize programmatic costs, while enhancing training benefits afforded to the soldier. Recognized as the Army's live training standard, the LT2 product line architecture, standards, assets, and common operating environment have been used by more than 16 major Army and Department of Defense live training programs with more than 130 systems fielded.

In addition, LT2's 2GPLE approach is exhibiting the following benefits:

- **More efficient integration of the Army products by the use of common standards and products to meet training and test requirements**
- **Compatibility of objective system and products with evolving capabilities**
- **Wider interoperability before executing subsystem and device production**
- **Reduced total lifecycle costs to include acquisition, development, testing, fielding, sustainment, and maintenance.**

This continuing transformation has generated a significant return on investment to date within PM TRADE's live training system acquisition portfolio. The first generation approaches generated more than $300 million in cost avoidance across the development of live training systems to include Combat Training Centers Instrumentation Systems, Home Station Instrumentation Systems, Instrumented Ranges, and Targetry. The second generation approach, known as Consolidated Product Line Management or CPM in the Army, is projected to save another $200 million over the next two to five years[2].

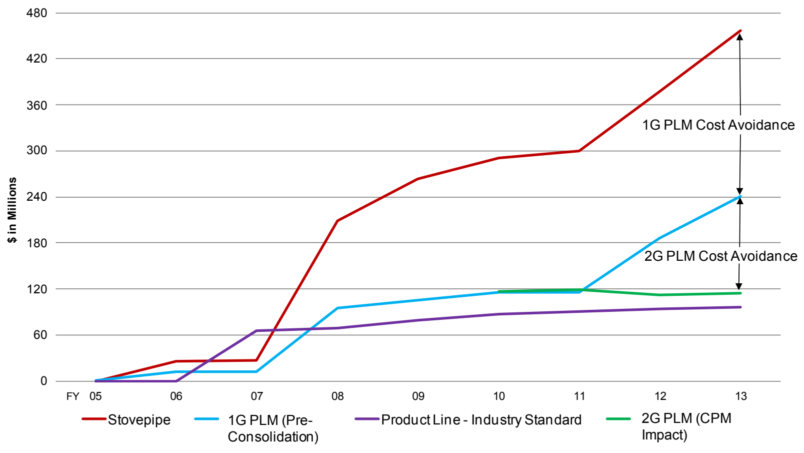


Figure 2. Cost avoidance benefits of product line engineering for LT2

2GPLE in the Navy: AEGIS Combat System

The AEGIS Combat System is an integrated warfare system deployed on some 100 naval vessels in the U.S. Navy and the navies of key allies across the globe. AEGIS is deployed on deep-water fleet ships, Littoral Combat Ships, and (more recently) U.S. Coast Guard National Security Cutters (NSCs). As the

Aegis Combat System Engineering Agent, Lockheed Martin's Maritime Systems and Sensors Division maintains the Common Product Line (CPL) requirements in a common DOORS database and source code in a Common Source Library (CSL) that is maintained for all product configurations, and they do it using the 2GPLE paradigm.

The primary objective of CPL is to develop once, and build and deploy many times from one set of common assets—principally requirements, source code, and tests. The AEGIS Baseline 9 Common Product Line comprises the requirements and source code that is maintained for all product configurations. CPL supports the US Navy's objective to more quickly field capability as well as the goal of minimizing cost and schedule for delivering computer program capability updates.

The CPL methodology is in high gear for the current AEGIS Baseline 9, which is the foundation for cruiser and destroyer platforms as well as Land Based Ballistic Missile Defense (BMD). The CPL approach enables the deployment of products from the combat system on the Littoral Combat Ships (LCS) and the US Coast Guard NSCs. It is also the basis for all future domestic and international AEGIS and LCS development efforts.

Figure 3. The Aegis destroyer USS Hopper (DDG 70) launches a missile to intercept a short-range ballistic missile. (U.S. Navy photo/Released)

CPL enables the critical convergence of AEGIS antiaircraft warfare and BMD functionality while providing the fleet with affordable capability and timely upgrades that keep pace with evolving threats. The CPL approach encompasses all phases of the classical V-chart. In the requirements development phase, requirements are consolidated into a single database (using IBM Rational's DOORS tool) for all stakeholder programs using Gears as the variation engine. This approach avoids redundant efforts and requirements capture when managing program-unique databases. Verification of the requirements is also maintained in the DOORS database.

In the software implementation phase, a master software development repository (CSL) is utilized that contains source files, libraries and configuration files that support multiple product configurations. Products comprise common and unique capabilities such that modifications to common configurations are implemented once and feature-based variation is used to automatically include or exclude each capability from a product.

LT2 SPREADS ACROSS THE SERVICES

The hundred-plus systems deployed as members of the LT2 family include these in the Air Force and Marines, as well as other commands within the Army:

CIEDAS—Counter Improvised Explosive Device (IED) After Action Review System (USAF)

The LT2 Homestation Instrumented Training System (HITS) product was heavily leveraged in creating the Air Force's CIEDAS product for convoy counter IED training. An early version of what became the Digital Range Training System (DRTS) Integrated Player Unit (IPU) was used to instrument Air Force convoy vehicles providing multiple in-vehicle video feeds and position/location information to the mobile Exercise Controller (EXCON). Temporary mobile field cameras provided additional video coverage. The LT2 product line HITS software components and Common Training Instrumentation Architecture (CTIA) provided the basis for exercise control, player unit monitoring and control, and After Action Review (AAR) reporting. Common software components provided the video monitoring and editing, and a new rapid AAR capability was developed that allowed an on-going exercise run and an after action review presentation simultaneously with a single operator.

SMS—Soldier Monitoring System (Army—SOCOM)

The Soldier Monitoring System provides safety monitoring of special forces students conducting a land navigation exercise. CTIA and HITS provide the foundation of the exercise control and AAR capabilities of SMS. The player unit radio instrumentation takes advantage of the standard LT2 Player Unit gateway, CTIA provides the architecture and event distribution mechanism, and HITS components provide situational awareness capabilities.

I-TESS II—Instrumented - Tactical Engagement Simulation System II (USMC)

I-TESS II provides the USMC with dismounted instrumentation in support of direct force-on-force tactical training. The LT2 HITS product was used in its entirety as the exercise command and control and after action review capability. Modifications to HITS were created to provide USMC customizations to support their unique style of training. These changes were approved by the LT2 Core Asset Working Group (CAWG) Integrated Product Team (IPT) and absorbed by the LT2 product line.

MC-ITS—Marine Corps Instrumentation Training System (USMC)

MC-ITS was a predecessor to RISCon that provided force-on-force tactical training for the USMC. HITS was used in its entirety as the foundation for this program. Specific new functionality was added to HITS to mainly support USMC IED training and specialized IEDs and IED jammers. The modifications produced by this program have just recently been rolled into the LT2 product line.

RISCon—Range Instrumentation System Control (USMC)

The RISCon program's objective is to reduce sustainment, operational, and enhancement costs of the existing and future Marine Corps Range Instrumentation System Product Line. RISCon leverages the CPM construct of tools (i.e. Gears) and processes to establish and manage a framework for affordable USMC Product Line operation, improvements and deployments. The project leverages the US Army's LT2 Product Line using CTIA. CTIA establishes the framework (protocols, standards, interfaces, etc.) for developing a repository of LT2 core components.

Work for Naval Air Systems Command (NAVAIR) and you'll support our Sailors and Marines by delivering the technologies they need to complete their mission and return home safely. NAVAIR procures, develops, tests and supports Naval aircraft, weapons, and related systems. It's a brain trust comprised of scientists, engineers and business professionals working on the cutting edge of technology.

You don't have to join the military to protect our nation. Become a vital part of NAVAIR, and you'll have a career with endless opportunities. As a civilian employee you'll enjoy more freedom than you thought possible.

Discover more about NAVAIR. Go to **www.navair.navy.mil**.

Equal Opportunity Employer | U.S. Citizenship Required

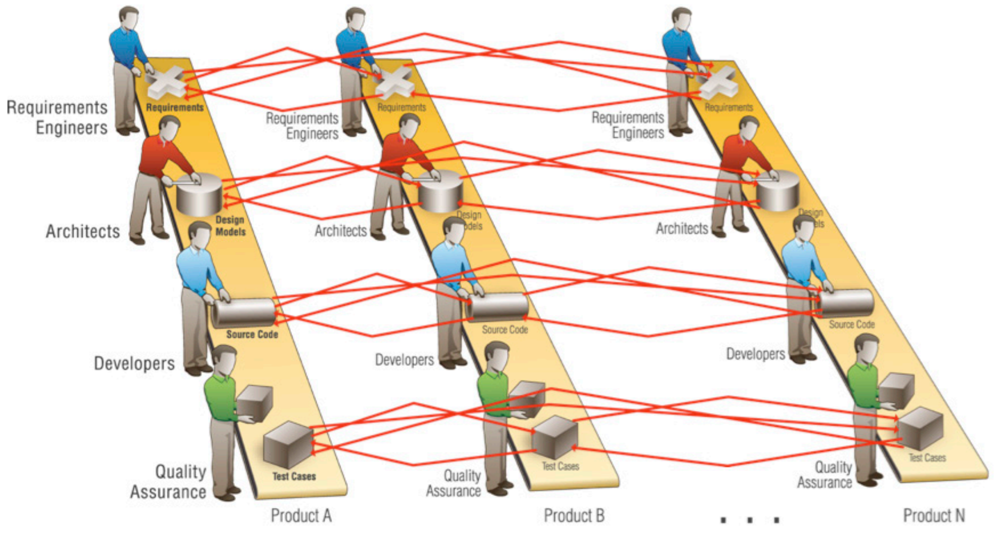


Figure 4. Product-centric development and O(N²) complexity

During the test and verification phase, CPL utilizes a consolidated testing approach to maximize efficiency of common requirements and capabilities. This results in tailored regression testing based on changed functional areas. This also utilizes an integrated test team using common test plans and procedures. Common test efforts are leveraged and consolidated problem reporting avoids duplicate reporting caused by redundant testing. These test benefits are currently being realized as AEGIS baseline 9 prepares for certification.

Organizational consolidation became possible under product line development. Overall program management was consolidated to minimize redundancy and achieve a common program structure and consolidated business rhythm, metrics, and reviews. An engineering product team was established that spans programs to maximize commonality and to drive consistency and design practices. An Engineering Review Board was established as a decision authority to ensure proper CPL behavior at the product level for each of the elements.

The benefits were highlighted when the US Coast Guard made the decision to enter the family with their new National Security Cutter. Once in the product line, they avoided the months it would have taken to implement and verify the hundreds of fixes and upgrades that set their application apart. Instead, the Coast Guard applied their unique feature-based requirements to the CPL DOORS database using Gears, and thus avoided having to apply the specification changes one by one. This resulted in a much quicker deployment of code and requirements for the Coast Guard.

Conclusion

Although this is primarily the story of an Army and a Navy program, LT2 and AEGIS have put down 2GPLE roots in every Service. Aegis has brought the Coast Guard into its product line family. And the hundred-plus LT2 family members include several developed for and in use by the Air Force and Marines.

There are organizational, management, and contracting issues that these programs have had to surmount, but their success shows that those issues are tractable. As a result, they would seem to provide strong evidence that Second Generation Product Line Engineering is an engineering discipline suitable for DoD acquisition programs, across Services and domains. Like its first-generation predecessor methods, it is showing multiple-integer improvements in quality, time to deployment, cost, and engineering productivity.

ABOUT THE AUTHORS

Dr. Paul Clements is the Vice President of Customer Success at BigLever Software, Inc., where he works to spread the adoption of systems and software product line engineering. He was previously at Carnegie Mellon's Software Engineering Institute, where for 17 years he worked in software product line engineering and software architecture documentation and analysis. Clements is co-author of three practitioner-oriented books about software architecture as well as the field's leading text on software product line engineering.

E-mail: pclements@biglever.com
Phone: 512-567-1681

Susan P. Gregg is a Principal Project Engineer for the Lockheed Martin Corporation. She holds a B.A. in Physics from Rutgers University. She has over 30 years experience is systems and software engineering. She is currently the Technical Director for the US Navy's Common Product Line.

E-mail: susan.p.gregg@lmco.com
Phone: 856-359-1636

Dr. Charles Krueger, BigLever founder and CEO, is a thought leader in the product line engineering field with 25 years of experience in software engineering practice and more than 60 articles, columns, book chapters, conference keynotes, and session presentations. Krueger has proven expertise leading product line development teams, and helping establish notable PLE practices in companies such as General Motors, Lockheed Martin, General Dynamics, Ikerlan/Alstom, and three Software Product Line Hall of Fame inductees.

E-mail: ckrueger@biglever.com
Phone: 512-426-2227

Jeremy T. Lanman, Ph.D. is the lead architect for the Common Training Instrumentation Architecture and Live Training Transformation Product Line at the U.S. Army PEO STRI. His professional experience includes 10 years of DOD acquisition and systems engineering of military simulation and training systems. Dr. Lanman received his B.S in Computer Science from Butler University, M.S. in Software Engineering from Embry-Riddle Aeronautical University, and Ph.D. in Modeling and Simulation from the University of Central Florida.

E-mail: jeremy.lanman@us.army.mil
Phone: 407-384-5307

Jorge Rivera is currently working for General Dynamics C4 Systems out of the Orlando facility leading live training efforts and supporting the 2nd Generation Product line instantiation under the CPM contract. His prior experience includes 25 years of DoD Acquisition service with over 15 years of those in the live training domain. As the Assistant Project Manager (APM) LT2, Mr. Rivera championed the LT2 product line and managed the CTIA & FASIT efforts. He earned his B.S. in Electrical Engineering (EE) from the University of Puerto Rico in 1983 and his M.S. in EE from Fairleigh Dickinson University, NJ in 1987

E-mail: jorge.rivera@gdc4s.com
Phone: 407-275-4820

Rick Scharadin has over eighteen years of Senior Program Management experience related to complex large scale system development, open architecture designs, software product line development, product integration, test and delivery for various Navy Aegis Baselines. He has accumulated over his career with Lockheed Martin 12 service awards, including manager of the year in 2001. Rick has a BS in Electrical Engineering from Penn State and a MS in System Engineering from Stevens Institute.

E-mail: richard.w.scharadin@lmco.com
Phone: 609-326-4685

James T. Shepherd works as a Lead Architect on the Aegis software common product line for Lockheed Martin MS2. He holds a B.S. in Computer Science from Montclair State University and an M.S. in Computer Science from Drexel University. He has more than 25 years experience in systems and software engineering of mission critical applications for the US Navy.

E-mail: james.t.shephard@lmco.com
Phone: 609-326-4685

Andrew J. Winkler is a Principal Engineer at Lockheed Martin and has over 15 years experience working on large scale systems, including the AEGIS Combat System and the DDG1000 C3I system. Most recently Andrew has held the role of System Architect for the US Navy's AEGIS Common Product Line. Andrew has a BS and MS in physics from the University of Vermont.

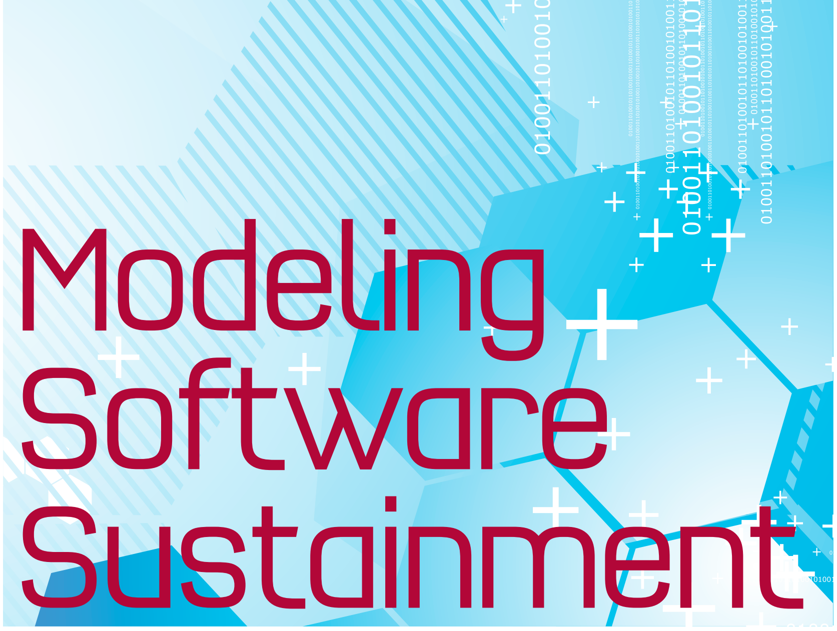
E-mail: andrew.j.winkler@lmco.com
Phone: 856-914-6318

REFERENCES

1. Bachmann, F., Clements, P. “Variability in Software Product Lines,” Technical report CMU/SEI-2005-TR-01, Software Engineering Institute, 2005.
2. Clements, P. and Bergey, J. The U.S. Army’s Common Avionics Architecture System (CAAS) Product Line: A Case Study, Technical Report CMU/SEI-2005-TR-019, September 2005.
3. Clements, P.; Northrop, L. Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.
4. Cohen, S., Dunn, E., Soule, A. , Successful Product Line Development and Sustainment: A DoD Case Study, CMU/SEI-2002-TN-018, September 2002.
5. Dillon, M., Rivera, J., Darbin, R., Clinger, B., “Maximizing U.S. Army Return on Investment Utilizing Software Product-Line Approach,” Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC), 2012.
6. FedSmith.com, “Billions Wasted…,” <http://www.fedsmith.com/article/313/billions-wasted-dod-because-duplicate-business-systems.html>
7. Flores, R., Krueger, C., Clements, P. “Mega-Scale Product Line Engineering at General Motors,” Proceedings of the 2012 Software Product Line Conference (SPLC), Salvador Brazil, August 2012.
8. Guertin, N., and Clements, P., “Comparing Acquisition Strategies: Open Architecture vs. Product Lines,” Proceedings of the 2010 Acquisition Research Symposium, Monterey, May 2010.
9. Jensen, Paul. (2009). “Experiences with Software Product Line Development.” CrossTalk 22, 1 (January 2009): 11–14.
10. Kang, K.; Cohen, S.; Hess, J.; Novak, W.; & Peterson, A. “Feature-Oriented Domain Analysis (FODA) Feasibility Study” (CMU/SEI-90-TR-021, ADA235785). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1990.
11. Krueger, C. “The Systems and Software Product Line Lifecycle Framework,” BigLever Software Technical Report #200805071r3, 2010. http://www.biglever.com/extras/SplLifecycleFramework.pdf.
12. Linden, Frank J. van der, Schmid, Klaus, Rommes, Eelco. Software Product Lines in Action, Springer, 2007.
13. Rivera, J., Samper, W., Clinger, B. (2008). Live Training Transformation Product Line Applied Standards For Reusable Integrated And Interoperable Solutions. Paper No. 483; MILCOM 2008.
14. Software Engineering Institute, “Catalog of Software Product Lines,” <http://www.sei.cmu.edu/productlines/casestudies/catalog/index.cfm>
15. SPLC Product Line Hall of Fame, <http://splc.net/fame.html>
16. UAV Forum, Librarian’s Desk, <http://www.uavforum.com/library/librarian.htm>

NOTES

1. This is an adaptation of the Software Engineering Institute’s definition of a software product line, which is a product line in which software plays a central role in the systems :3]
2. These figures are based on industry standard estimates of code cost, and are calculated assuming that post-deployment software support constitutes 70% of development cost and a life expectancy of 10 years. See [5] for a more detailed explanation.



Modeling Software Sustainment

Robert Ferguson, SEI
Mike Phillips, SEI
Sarah Sheard, SEI

Abstract. Software sustainment is critical to DoD capability, but it is difficult to determine where and when to invest limited funds to produce the most significant impact for the least amount of effort and expense. An SEI research initiative is developing a model that shows the results of various investment decisions, allowing decision makers to see the effects and make adjustments before problems occur. Determining what data are needed to make the model work and how to collect it is also a significant challenge addressed by this research.

Introduction

Over the years, the percentage of functionality that depends on software has increased rapidly, making the cost of sustaining that software grow exponentially. For example, in the armed services the number of weapon system platforms is diminishing, but their projected service lifetimes are expanding. The B-52 now has a planned 90-year lifetime, and it includes functionality that could never have been imagined by its designers in the late 1940s. While hardware sustainment typically focuses on maintaining structural integrity, software sustainment is what continues to grow the capability of the B-52 and many other platforms like it. Even in the face of technological uncertainty, sustainment organizations across the DoD must plan for—and sustain—their capability to continuously improve critical software-intensive systems, update after update.

The SEI has worked with software sustainment groups in each of the services to determine how to make the best use of their dedicated software resources. In response, we are developing an economic model that can be used by decision makers to determine where and when to invest to have the greatest impact on long-term costs and fleet readiness.

Background

CrossTalk has been documenting the issues surrounding sustainment for several years. In the December 2007 issue, Capers Jones pointed out 24 major reasons that software in aging systems must be "improved" [1]. (Whether this would be called "maintenance" or "sustainment engineering" was a sidebar addressed by the editor of CrossTalk at that time, Beth Starrett.) In the same issue, the future challenges of sustaining F-35 software were described by Lloyd Huff and George Novak [2].

These future challenges led to two related studies by the Air Force in 2011. An Air Force Science Advisory Board report, "Sustaining Air Force Aging Aircraft into the 21st Century" [3] noted that sustainment was an inherently expensive process that would eventually involve the remanufacture of the entire aircraft, component-by-component, as wear-out occurred. Significant technical challenges to this type of sustainment effort were recorded as failure modes became age-driven rather than usage-driven. Significant concern was expressed in the report: "The Air Force is concerned that the resources needed to sustain its legacy aircraft may increase to the point where they could consume the resources needed to modernize the Air Force." The report sought to identify key technologies that could reduce the time and expense for the Air Force sustainment enterprise in its quest to maintain and field these aircraft through the 21st century.

The second report is the Air Force Studies Board's "Examination of the U.S. Air Force's Aircraft Sustainment Needs in the Future and Its Strategy to Meet Those Needs" [4] which addresses the broad issues of sustainment, with a specific chapter on software challenges.

The sustainment problem is made more complex because the funding decisions involve an understanding of the tensions among three different perspectives with differing definitions of value: operational need (warfighter view), the management of the portfolio (materiel view), and the capability and capacity of the sustaining organization (process, skills, tools, and people). DoD leaders must make decisions about allocating resources between the efforts that support the warfighter and the efforts that improve the performance of the sustainment organization consistently, with the goal of optimizing long-term value to the services.

The economic model that our research initiative is developing to support decisions about these investment questions will analyze factors such as demand for sustainment, the capacity of an organic workforce to do the sustainment, and the timing of funding, in terms of its impact on long term costs and the readiness of aircraft fleets. As part of this work, we developed an initial model that shows the interaction of the stakeholder values and the allocation of investment as a systems dynamics (or time-based) model. This type of model uses stocks and flows to represent sustainment performance over time [5].

Foundations of Our Approach

Systems dynamics work traces its roots to Jay Forrester at MIT in the 1950s and has been used as a modeling approach in the study of economics and organizations. Systems dynamics models allow people to study systems with many interrelated factors. When many factors are changing at once, their interaction can cause emergent effects that can result in a sudden and dramatic change in outcome. For these situations, traditional, simpler economic models such as return-on-investment and net present value are insufficient to understand what is happening. Through the modeling and analysis research, we are looking for the minimum amount of data, a signal, that can forecast a sudden and dramatic change (a "tipping point"). Forecasting the tipping point gives decision makers time to take action before a problem becomes intractable. Our research asks the following questions:

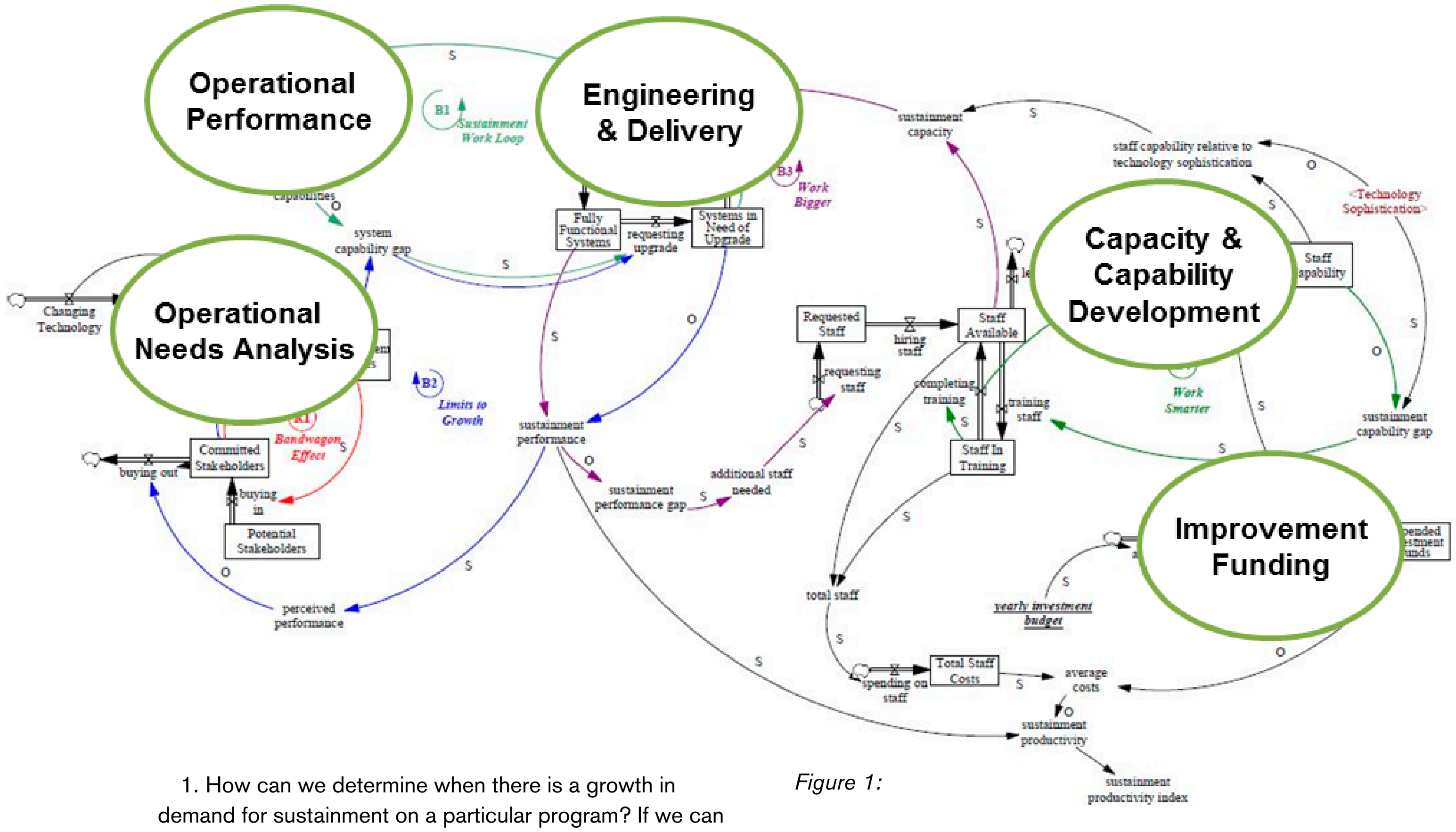


Figure 1:

1. How can we determine when there is a growth in demand for sustainment on a particular program? If we can identify the needed data, will it be possible to collect it and do the necessary analysis?

2. Do the models we are developing provide actionable information to decision makers in a timely manner? For example, does reallocating some resources from the sustainment work to the development of the workforce (tools, etc.) help reduce the cost of sustainment?

3. What measure of warfighter readiness correlates to the predictive factors in the model?

Using this approach, we aim to help DoD programs better plan their financial investment in software sustainment to ensure that the products are sustainable for as long as possible and deliver the best value for the taxpayer dollar. A working model will help senior managers test alternative strategies for investment. Satisfying the senior managers that the model correctly anticipates the behavior of the systems requires us to focus our research on discovering what sources of data can be used to calibrate the model for real application. Accomplishing this requires us to:

- **identify data collection points within the sustainment processes**
- **identify opportunities to measure warfighter readiness or system use**
- **develop standards for applying data collection across different sustaining organizations**

The Systems Dynamics Model

The basic goal of a simulation model is first to represent the normal behavior of a system and then to introduce a new input to see how the responses change. The model that we have developed represents the behavior of the different aspects of the sustainment process, including the warfighter, the technical capability of the sustainment organization, and the capacity of the sustainment organization to deliver the work. We are testing the system response to various change scenarios, including the following:

Threat. An external change (such as a new threat to the warfighter) results in a request to update the system capability. This request means the sustaining organization will have to perform both product and process changes; the development process and testing may need to change as well. The changes often require funding to re-equip the facility and re-train the workforce. Our systems dynamics model helps decision makers analyze the effect if funding for this improvement is delayed.

Support Technology. The sustainment organization decides to improve its own throughput and adopts new processes to "do more with less." Typically the change is also in response to new quality goals. In this case our model helps codify the effect on sustainment capability, and capacity and therefore on operational performance.

Workforce. Sequestration effectively decreases the staff available to sustainment organizations by 10% to 20 %. How does this decrease affect a sustaining organization's ability to meet its sustainment demand? Does it affect aspects of the warfighter mission as well?

Our current model of sustainment consists of five basic processes and five dynamic feedback loops, shown in Figure 1. (Model details are blurred to emphasize the loops rather than exact feedback forms.) Process definitions provide suggestions for measures of inputs and outputs. Those inputs and outputs can then be calibrated for the forces and feedback functions.

The processes in the model are listed below with input and output suggestions:

1. Operational Performance
Input: Missions measured by capabilities used and mission-capable availability
Output: Action reports measured by %success, and availability gap

2. Operational Needs Analysis
Input: Mission performance measures and new potential threats, technologies, uses, and mission-capabilities
Output: New capability definition

3. Engineering & Delivery
Input: Sustainment demand (accepted and not-accepted requests)
Sustainment capability required (skills, tools, facilities)
Sustainment capacity required (throughput)
Output: Delivered products by count of deployments and costs
Sustainment gap (requests not accepted)

4. Capacity & Capability Development
Input: Changes to training, tooling, facility, processes
Hiring, furloughs, and attrition
Output: Capacity available (%of request)
Capability available date or delay

5. Improvement Funding
Input: Funding requested for capability and capacity development
Output: Time required to fund, amount funded

The following dynamic loops have been identified:

1. Bandwagon Effect. Successful missions and high mission performance lead to additional demands for capacity and capability.

2. Sustainment Work. Product use and environmental effects increase demand for sustainment work.

3. Limits to Growth. Capacity and capability of a sustainment organization limit the rate of completion of sustainment work. As these limits begin to extend the time required to redeploy, the long-term effect may be a reduction in demand or a switch to an alternate platform.

4. Work Bigger. A sustainment organization may attempt to meet sustainment demand by requiring overtime work or employing extra contract employees. Either of these approaches may work for a short time or a small additional cost, but they stress the organization and quickly reach the limits of their effectiveness. The organization can hire staff, but it must also allow time for training and acculturation of new hires to meet performance objectives.

5. Work Smarter. A sustainment organization invests in new capabilities (skills, tools, and processes) and possibly additional resources (people and facilities) to improve capacity for sustaining work.

Each of these scenarios entails several decisions in the process loops and stimulates response curves from the model. The response curves help decision makers forecast how deferring decisions or reallocating resources affects both warfighters and sustainment organizations. Our systems dynamics model will be helpful to decision makers if they are able to make faster decisions and if the data from the model makes it easier to get sponsor support for the decisions. Our systems dynamics model will be helpful to researchers because the inputs/outputs of the model process suggest how to validate the model with sustainment data.

Initial Results and Future Work

Our research thus far has shown that the system dynamics model exhibits the expected and observed behavior of product sustainment. Of particular interest has been the impact of workforce changes on the sustainment cycle—positive effects like training and new software tool sets, and negative ones like the recent furloughs. As the system responds to change, the model helps us see that the effects have distinct cycles. A furlough has immediate impact, but a decision to fund new engineering environments can take years.

Model calibration is needed to capture specific, real world situations. Calibration will require significant work with programs and sustainment organizations. We are initiating a collaboration with a sustainment organization that could potentially provide the needed data for this research, and we are soliciting broader participation across the services and agencies.

Additional Reading:

- McGarry, J. Software Maintenance Life Cycle Cost Estimation Model. Proc. of the PSMSC User Group, Portsmouth, Virginia, 2012. <http://www.psmsc.com/UG2012/Workshops/w4-%20files.pdf>
- Rosser, J. B. From Catastrophe to Chaos: A General Theory of Economic Discontinuities: Mathematics, Microeconomics and Finance (Vol. 1). Kluwer Academic Pub, 2000.
- Jones, C. "The Economics of Software Maintenance in the Twenty-first Century." ComputerAid Inc. (CAI), 2006. <http://www.compaid.com/caiinternet/ezine/capersjones-maintenance.pdf> ❖

REFERENCES

1. Jones, C. "Geriatric Issues of Aging Software." 20.12 CrossTalk (December 2007): 4-8.
2. Huff, Lloyd and Novak, George. "Performance-Based Software Sustainment for the F-35 Lightning II." 20.12 CrossTalk (December 2007): 9-14.
3. United States Air Force Scientific Advisory Board, "Sustaining Air Force Aging Aircraft into the 21st Century" (SAB-TR-11-01). August 2011.
4. National Research Council. "Examination of the U.S. Air Force's Aircraft Sustainment Needs in the Future and Its Strategy to Meet Those Needs." National Academies Press, 2011.
5. Sterman, J.D. Business Dynamics: Systems Thinking and Modeling for a Complex World. Irwin/McGraw-Hill, 2000.

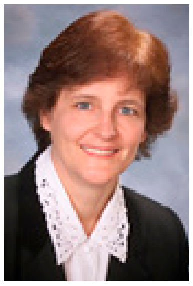
ABOUT THE AUTHORS

Robert Ferguson is a senior member of the technical staff at the Software Engineering Institute. He works primarily on software measurement and estimation. He spent 30 years in industry as a software developer and project manager before coming to the SEI. His experience includes applications in real-time flight controls, manufacturing control systems, large databases, and systems integration projects. He has also frequently led process improvement teams. Ferguson is a Senior Member of IEEE and has a Project Management Professional (PMP) certification from the Project Management Institute (PMI).

4500 Fifth Ave
Pittsburgh, PA 15213
E-mail: rwf@sei.cmu.edu
Phone: 412-268-9750
Fax: 412-268-5758

D. Mike Phillips is a principal research engineer at the Software Engineering Institute. He led a team that created the CMMI Product Suite, successfully describing key practices for both systems and software engineering. As an Air Force senior officer, Phillips led an Air Force program office's development and acquisition of the software-intensive B-2 Spirit stealth bomber using integrated product teams. He holds a B.S. in astronautical engineering from the U.S. Air Force Academy, an M.S. in nuclear engineering from Georgia Tech, an M.S. in systems management from the University of Southern California, an M.A. in international affairs from Salve Regina College and an M.A. in national security and strategic studies from the Naval War College.

4500 Fifth Ave
Pittsburgh, PA 15213
E-mail: dmp@sei.cmu.edu
Phone: 412-268-5884
Fax: 412-268-5758



At the Software Engineering Institute, Dr. Sheard researches software engineering process and measurement and brings software engineering tools and technologies to government clients. Previously she was a consultant and teacher at Third Millennium Systems and at the Systems and Software Consortium, and a systems engineer at Loral/IBM Federal Systems and Hughes Aircraft Company. She has a Ph.D. in Enterprise Systems from the Stevens Institute of Technology, a master's degree from the California Institute of Technology, and a bachelor's degree from the University of Rochester.

4500 Fifth Ave
Pittsburgh, PA 15213
E-mail: sheard@sei.cmu.edu
Phone: 412-268-7612
Fax: 412-268-5758

Using Combinatorial Testing to Reduce Software Rework

Redge Bartholomew, Rockwell Collins

Abstract. In developing many safety-critical, embedded systems, rework to fix software defects detected late in the test phase is the largest single cause of cost overrun and schedule delay. Typically, these defects involve the interactions among no more than 6 variables, suggesting that 6-way combinatorial tests could detect them much earlier. NIST developed an approach to automatically generating, executing, and analyzing such tests. This paper describes an industry proof-of-concept demonstration to see if this approach could significantly reduce the number of defects that escape into the test and evaluation phase of safety-critical embedded systems.

Introduction

Studies of safety-critical, embedded systems have shown that the rework required to fix late-detected software defects is one of the largest single components of their development cost and schedule—e.g., [1][2][3][4][5]. They also show that detection of these latent defects accelerates during late-stage testing and that those detected during operational test and evaluation have become more than just problematic. Much of this is attributable to verification tools and techniques that are becoming increasingly inadequate as the scale and complexity of software continues to increase [6][7][8][9]. An emerging need to develop parallel software for embedded multicore processors will make this problem worse [10]. Improvement requires tools and methods that prevent defect injection or that accelerate detection. They must do so, however, without a prohibitively large impact on normal development.

A study conducted by NIST and NASA looked at software defects detected over a 15-year period [11]. Systems studied included avionics, medical devices, web browsers, servers, space systems, and network security systems, and ranged in size from tens of thousands to hundreds of thousands of lines of code. It found that defects were triggered by the interactions among no more than six variables. This being the case, 6-way combinatorial test vectors might be able to detect them. Subsequently, NIST and the University of Texas-Arlington found an efficient algorithm for minimizing the number of test vectors that would cover up to 6-way combinations of input values [12][13][14][15]. They implemented this algorithm in a tool called Automated Combinatorial Test System (ACTS)[1].

The tool was effective at triggering defects, but verification testing required expected outputs, not just inputs, and creating these manually for thousands of inputs would be prohibitively expensive. NIST found an approach to automating this process using a model checker's counter examples. It also created a utility that merged the input vectors with their expected outputs as well as a test harness that read complete test cases, executed tests, analyzed results (compared actual versus expected outputs), and identified anomalies [16].

This paper describes an industry proof-of-concept study that used NIST's approach to automate unit testing of a software defined radio's control software. The goal was to determine if the NIST approach could cost-effectively reduce the number of latent software defects escaping into system testing and at the same time achieve the structural coverage required by regulatory authorities.

The Test Environment

Tests were generated, executed, and analyzed on a Windows 7, quad-core, 2.5 GHz, i5 laptop with 4GB memory. ACTS was used to create a model of the input variables, generate 6-way combinatorial test vectors, and export them to a networked server. The NuSMV[2] model checker generated the state space and exported it to the same networked server. An in-house utility function read the two files, searched the state space for states containing the ACTS vectors, reformatted them, and exported them as test cases back to the server. A commercial test harness, VectorCAST, instrumented the source code to track structural coverage, measured code complexity, imported the test case file, loaded test values into input variables, and executed tests. It also accumulated the achieved modified condition/decision coverage (MC/DC) [18], collected output variable values, compared actual with expected values, and identified discrepancies.

The code being tested was a software defined radio's control interface, containing 196,000 executable source lines of C++ code. The initial focus of the study was a code unit responsible for controlling the radio's waveform mode (e.g., HAVEQUICK, SINCGARS, Link 4) and operational state (e.g., idle, ready, running). This had 579 lines of code, 34 input variables, and 4 output variables of interest, used by 47 decisions nested up to 8-levels deep, spread over a 6-case switch. Its measured complexity (number of unique execution paths) was 46. In addition to the mode and state controller, the study tested another 70 of 717 code files.

Defining the Input Space

Developers provide ACTS with a name, a data type, and a set of values for each input variable. They also select the combinatorial strength of the vector generation (2-way through 6-way). ACTS then generates a set of input vectors containing all combinations of input variable values for the selected strength. Table 1 shows the 2-way vectors ACTS generated for the function:

```
if (c == true)
    e = a + b;
else
    e = a * d;
return e;
```

	a	b	c	d
1	0	255	true	-1
2	0	256	false	0
3	0	255	false	1
4	15	256	true	-1
5	15	255	true	0
6	15	256	false	1
7	16	255	false	-1
8	16	256	true	0
9	16	255	true	1

Table 1: Two-Way Combinatorial Vectors

Defining the input space to maximize defect detection and structural coverage without significant test iteration (test, measure coverage, determine coverage gaps, add input vectors, repeat) is nontrivial [12]. The greater the number of input test values, the greater the code coverage but also the greater the likelihood of combinatorial explosion. The smaller the number, the greater is the likelihood of missed defects and inadequate structural coverage.

A compromise is to limit input values to those representing equivalence classes [16]. For each input variable, possible values are segregated into groups that would ostensibly produce no difference of interest in code behavior or output value. One or more representative values are then picked from each group. This typically includes values that test behavior across instruction and memory architecture boundaries (e.g., positive and negative minimum and maximum values, and 0), data definition ranges, coordinate systems, units of measure, and so on, and also those that drive decision conditions.

Identifying representative values for boundary values was straightforward. Finding values for condition variables in complex, nested logic—values that would force the execution paths required for code coverage—took more time. MC/DC requires that every condition in a decision has taken all possible outcomes at least once, and that each condition in each decision has been shown to independently affect that decision's outcome. Demonstrating independence-of-outcome typically requires modifying each condition in a decision while all others remain fixed, and showing that this modification has changed the outcome of the decision. For the while-loop in

```
if ((a != b) && (a != c))
{
  ...
  while ((a != b) && (a != c))
  {
    a = chan ();
  }
}
```

tests must be run to show that when both conditions are true, the loop is executed, and that when each is false but the other true, the loop is not executed. To determine the input space, values that force execution of each such path under the required conditions must be selected for each variable of each condition of each decision.

Enabling those values was difficult when the condition variable was an input and the values had to be loaded by an external procedure invoked from within a decision. In the example, the loop decision must be tested when $a = b$ and when $a = c$, neither of which conditions can be created by direct input from a test case. The value of *a* must be changed at runtime by the call to the external procedure *chan ()*, which is stubbed-out for unit test. The work-around was to add test-unique variables to the test cases generated by ACTS and the model checker. Test stubs were replaced with small procedures that loaded the value of the test-unique variable directly or indirectly into the condition variable. In the example, the test variable's value would be loaded into the return value of *chan ()*.

Generating a state space for all 34 input variables of the mode-state controller produced combinatorial explosion. Several separate sets of test vectors had to be generated instead, each set covering only those variables that interact to produce an output. The test harness assigned default values to those variables not included in a test case. Maximizing structural coverage required running all such sets of tests. In no case, however, was there an output value affected by interactions among more than six input variables, and in aggregate all 6-way combinations of interacting variables were tested.

Generating Expected Outputs and Executing Tests

The model checker is given a model containing variable definitions, their relationships, their values in an initial state, and how their values are determined in subsequent states. It then generates the state space (or a binary decision diagram of it), each state mapping a combination of input variable values to output variable values. See Fig. 1 showing the mapping of the input values from Table 1 to the output variable, *e*. For all states in which the value of *c* is *true*, the value of *e* will be equal to the value of *a* plus the value of *b*, which is expressed as *c = true : a + b*. In all other states, the value of *e* will be equal to the value of *a* times the value of *d*, expressed as *TRUE : a * d*. Fig. 1b shows a segment of the generated state space—the value of *e* followed by the input values that produced it.

In the NIST approach, the process of creating expected outputs for an input test vector relies on a model checker's counter-examples [17]. Ordinarily, to verify requirements or a design, developers using a model checker would create a model like the one in Fig. 1a , but they would also write properties the model must preserve—e.g., there must always be a way for the variable e to be 0, there must always be a way for it to be 272. The model checker attempts to prove that the model preserves these properties. Where it finds a violation of a property (a counter-example—e.g., an execution path in which e can never be 0), it produces a trace of the states that led to the violation.

To have a model checker determine an expected output for a given input vector, developers could negate a property and use the counter example to trace back to the input values that produced it. For example, they could specify that the variable e must never be 0. The model checker would detect a state that violated this property and generate a counter example showing the state transitions from the initial input values (the input vector) to the point at which e became 0. A simple utility could create a complete test case from a counter-example by merging the value of the output variable with the values of the input variables that produced it [16].

This study used a slightly different approach, requiring a smaller learning curve. Instead of searching through counter examples generated by the model checker, the utility function searches for each input vector across the entire state space generated by the model checker. The model in Fig. 1a generated 36 states: those containing all possible combinations of variable values. As shown in Table 1, all 2-way combinations of inputs can be covered by the nine input vectors generated by ACTS. The utility function finds state 4 containing the input vector, {0,255,false,1}, eliminates any irrelevant inputs and outputs from the state, reformats the remainder (the input vector and its expected outputs), and exports the result, {0,0,255,false,1}, to the test harness. When it has found and exported all 9 test cases, it is finished.

Developers then load the test harness with both the source code and the test cases, and map the test case entries to input and output variable names—e.g., map the first entry of the input

```
MODULE main
VAR
 a : {0,15,16};
 b : {255,256};
 c : {true,false};
 d : {-1,0,1};

DEFINE
 e :=
  case
    (c = true) : a + b;
    TRUE : a * d;
    esac;
```

Fig. 1a. NuSMV Model

```
------- State    4 ------
  e = 0
  a = 0
  b = 255
  c = false
  d = 1
------- State    5 ------
  e = 272
  a = 16
  b = 256
  c = true
  d = 1
------- State    6 ------
```

Fig 1b. State Space Segment

test case in Fig. 1b (0) to the source code variable e, the second entry (0) to the variable a. They can then execute the tests. Failures and the achieved code coverage can be monitored in test harness windows. Correctness of the expected outputs (verifying the oracle) is established when the resulting test cases are able to detect all seeded defects with no false positives.

Results

Putting aside defective or incomplete requirements, misinterpretations of requirements and design decisions, and other errors not revealed by exercising the code, at issue was whether such an automated test approach could cost effectively detect all (or nearly all) implementation defects. Evaluation criteria included accuracy, structural coverage, scalability, execution time, maturity, ease of learning, and ease of use.

Accuracy was measured in two ways: as the percent of seeded defects the tests detected; and as the percent of false detections (number of false positive detections as a percent of total detections). Defects were manually and arbitrarily seeded into versions of the code by changing values in arithmetic and logic statements, changing arithmetic signs, reversing and negating comparisons, deleting statements, and so on. In all, there were over 200. After debugging the NuSMV model, the search-export utility, and the test harness definition, the generated tests triggered all defects with no false detections.

The initial set of tests achieved 75% statement coverage, 71% branch coverage, and 68% MC/DC. The relatively low initial coverage was the result of the inadequately defined input space, described earlier. With a better understanding of how the input space was to be defined, the subsequently generated test cases achieved 100% MC/DC.

Scalability was an evaluation of both size (in this case, the number of input and output variables) and logical complexity. As mentioned earlier, after limiting inputs to only interacting variables, test generation never again produced state space explosion. After using test variables to deal with loops that changed the value of their condition variables, there were no further complexity issues.

Execution time was acceptable: for the largest vector generation model (19 input variables, 1 output variable), ACTS produced 2775 input vectors in six seconds, NuSMV generated the state space in about 60 minutes, and searching it and building the test cases took just over eight minutes. The test harness imported them in 15 seconds, created their executable tests in 12 seconds, and executed and analyzed them in under eight minutes.

Cost effectiveness was a measure of the value-in-use (accuracy, coverage, scalability, and performance), the effort required to learn the approach, and the effort required to use it on an ongoing basis. Learning to use ACTS was simple. NIST provides a tutorial that takes about two hours to process and contains everything needed to begin using the tool. Initial definition of the 34 input variables used by the mode controller took four hours, including initial equivalence class determination and value selection. Using the .pdf tutorial from the NuSMV web site, learning to develop NuSMV models and to use the NuSMV simulator to generate the state space took 20 hours. After encountering state space explosion, generating sets of input vectors for only interacting variables and selecting equivalence class values to achieve 100% branch coverage took an additional 16 hours. Finding a way of achieving 100% MC/DC coverage without manual intervention took another 16 hours. In total, the learning curve was 84 hours. As errors were found in models, the worst-case time spent completely regenerating and re-executing tests was under 90 minutes, but more commonly was less than 15 minutes.

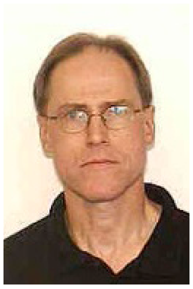
Maturity was an evaluation of readiness for deployment across a potential population of several thousand engineers—e.g., if the tools crash frequently or if they produce inconsistent, incorrect, or confusing results. The study used the 9-level NASA/DoD Technology Readiness scale[3] and found the toolset to be at Level 7, “System Prototype Demonstrated in [an operational environment]”. In summary, prototype software exists and all key functionality is available for demonstration or test; the tools were well integrated with operational systems; operational feasibility was demonstrated and most of the software bugs have been eliminated; and at least some documentation is available. A general deployment would require level 9 “Actual system [performance] proven through successful [developmental use].”

Conclusion

For unit test, this appears to be much more effective than the standard manual, iterative approach of writing tests, running them, checking coverage, writing more tests to fill coverage gaps, running more tests, and so on. Defining the input space to achieve required coverage consumed the largest amount of time, requiring several iterations of test case generation – especially to achieve full MC/DC. With experience, however, the number of iterations was significantly reduced. The study used staff with significant experience, but in general the approach required no knowledge or skills that could not easily be learned by an above average entry-level software engineer—e.g., creating and debugging the test generation models was much easier than writing and debugging the source code being tested.

Overall, results of the study were positive, although there are remaining issues of deployment packaging and tool licensing, training, mentoring, and technical support. Data for an empirical comparative evaluation of defect detection capability between combinatorial testing and other approaches do not exist, but there is enough evidence from literature to justify a pilot project or a trial deployment in a business unit. This is the current plan going forward.

ABOUT THE AUTHOR



Redge Bartholomew is with Rockwell Collins, currently researching tools and methods for automating the development of embedded software and for reducing the number of latent software defects found during test and evaluation.

400 Collins Road
M.S. 108-265
Cedar Rapids, Iowa 52498
Phone: 319-295-1906
E-mail: rgbartho@rockwellcollins.com

NOTES

1. The ACTS executable is available from NIST. See <http://csrc.nist.gov/groups/SNS/acts/index.html>
2. NuSMV is available from <http://nusmv.fbk.eu>
3. Technology Readiness Calculator at <https://acc.dau.mil/CommunityBrowser.aspx?id=320594&lang=en-US>

REFERENCES

1. Government Accountability Office, "F-35 Joint Strike Fighter: Current Outlook Is Improved, but Long-Term Affordability Is a Major Concern, GAO-13-309", March 2013
2. Government Accountability Office, "KC-46 Tanker Aircraft: Program Generally Stable but Improvements in Managing Schedule Are Needed, GAO-13-258", February 2013
3. Government Accountability Office, "Airborne Electronic Attack: Achieving Mission Objectives Depends on Overcoming Acquisition Challenges, GAO-12-175", March 2012
4. Jones, "Software Quality and Software Economics", SoftwareTech News, April 2010
5. Dvorak (ed.), NASA Study on Flight Software Complexity, March 2009.
6. National Academy of Sciences, Critical Code: Software Producibility for Defense, 2010
7. Baldwin, DoD Software Engineering and System Assurance, NDIA Proceedings of the 11th Annual Systems Engineering Conference, October 2008.
8. Afzal, Torkar, Feldt, Search-Based Prediction of Fault-Slip-Through in Large Software Projects, IEEE Symposium on Search Based Software Engineering, September 2010.
9. Andersin, TPI – a Model for Test Process Improvement, Seminar on Quality Models for Software Engineering, U of Helsinki, October 2004
10. Lu, Park, Seo, Zhou, Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics, ACM Proceedings of the 13th Annual International Conference on Architectural Support for Programming Languages and Operating Systems, March 2008
11. Kuhn, Wallace and Gallo, "Software Fault Interactions and Implications for Software Testing," IEEE Transactions on Software Engineering, June 2004.
12. Borazjany, Yu, Lei, Kacker, Kuhn, Combinatorial Testing of ACTS: A Case Study, Proceedings of the International Conference on Software Testing, Verification, and Validation, April 2012
13. Lei, Kacker, Kuhn, Okun, Lawrence, IPOG: A General Strategy for T-Way Software Testing, IEEE Proceedings of the Conference and Workshops on the Engineering of Computer-Based Systems, March 2007.
14. Kuhn, Lei, Kacker, "Practical Combinatorial Testing: Beyond Pairwise," IEEE IT Pro, May/June 2008.
15. Kuhn, Okun, Pseudo-Exhaustive Testing for Software, NASA/IEEE Proceedings of the 30th Software Engineering Workshop, April 2006.
16. Kuhn; Kacker; Lei, Practical Combinatorial Testing, NIST Special Publication 800-142, National Institute of Standards & Technology, October 2010.
17. Ammann, Black, Majurski, Using Model Checking to Generate Tests from Specifications, IEEE Proceedings of the 2nd International Conference on Formal Engineering Methods, December 1998.
18. RTCA, DO-178C: Software Considerations in Airborne Systems and Equipment Certification, RTCA, 2011.

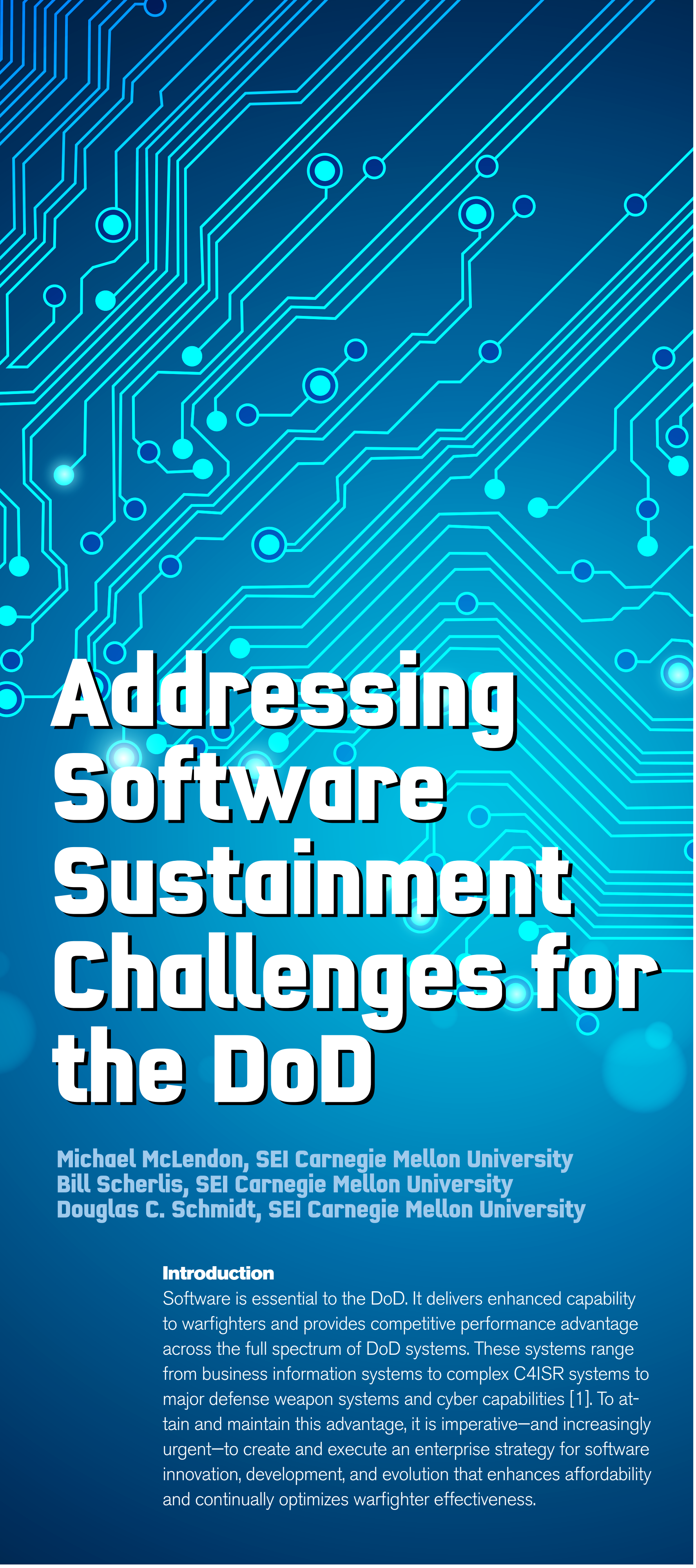
U.S. DEPARTMENT OF HOMELAND SECURITY

Homeland Security

The Department of Homeland Security, Office of Cybersecurity and Communications (CS&C) is responsible for enhancing the security, resiliency, and reliability of the Nation's cyber and communications infrastructure and actively engages the public and private sectors as well as international partners to prepare for, prevent, and respond to catastrophic incidents that could degrade or overwhelm these strategic assets. CS&C is seeking dynamic individuals to fill critical positions in:

- Cyber Incident Response
- Cyber Risk and Strategic Analysis
- Networks and Systems Engineering
- Computer and Electronic Engineering
- Digital Forensics
- Telecommunications
- Program Management and Analysis
- Vulnerability Detection and Assessment

To learn more about the DHS, Office of Cybersecurity and Communications, go to www.dhs.gov/cybercareers. To apply for a vacant position please go to www.usajobs.gov or visit us at www.DHS.gov.



Addressing Software Sustainment Challenges for the DoD

Michael McLendon, SEI Carnegie Mellon University
Bill Scherlis, SEI Carnegie Mellon University
Douglas C. Schmidt, SEI Carnegie Mellon University

Introduction

Software is essential to the DoD. It delivers enhanced capability to warfighters and provides competitive performance advantage across the full spectrum of DoD systems. These systems range from business information systems to complex C4ISR systems to major defense weapon systems and cyber capabilities [1]. To attain and maintain this advantage, it is imperative—and increasingly urgent—to create and execute an enterprise strategy for software innovation, development, and evolution that enhances affordability and continually optimizes warfighter effectiveness.

This enterprise DoD strategy must recognize the extent to which :

- Mission effectiveness depends on the ability of software developers and teams to deliver capability affordably and support the continual adaptation and enhancement of that capability
- Great value is provided to warfighters by enabling software-intensive functionality across the lifecycle so systems can operate interdependently and dependably in net-centric and cyber environments

It is hard to achieve these goals, however, due to rapid changes in mission environments and technology infrastructure, along with a challenging fiscal environment.

As DoD systems continue to age [2]—and sequestration and other budget constraints and uncertainties place greater emphasis on efficiency and productivity in defense spending [3]—it is increasingly important to create more efficient and effective approaches to sustaining and advancing the competitive edge that software provides. Software sustainment involves coordinating the processes, procedures, people, information, and databases required to support, maintain, and operate software-reliant aspects of DoD systems [4]. This article summarizes key software sustainment challenges faced by the DoD and highlights key R&D activities needed to address these challenges.

Software Sustainment Trends and Challenges

The software acquisition process delivers operational performance to meet identified warfighter requirements. Henceforth, systems transition into the sustainment phase. During sustainment, software-engineering processes and practices are continuously applied to (1) assure the ongoing competitive military advantage of a system and (2) ensure its seamless operation in helping to evolve net-centric and cyber infrastructures and environments. Various trends shape DoD policies and infrastructure for sustaining software, including:

- rapid performance advances associated with Moore's Law and associated hardware innovations (cost and capacity for storage, processing, and communications, and the consequent influence on computing systems architectures) that accelerate technology refresh cycles,
- the ever-increasing connectedness of systems, in which each system becomes a node in a vast, complex information network,
- the prevalence of closed-source and open-source off-the-shelf software technologies and practices, which commoditizes the market for software engineers with modern skills but creates gaps for projects that need staff with expertise in older technologies,
- the need to adapt software to address diminishing manufacturing sources stemming from the loss of producers or suppliers of hardware used in DoD systems,
- the challenges of modernizing and recapitalizing legacy DoD systems in a constrained budget environment that increasingly emphasizes greater efficiency and productivity in defense spending,
- the repurposing of systems to meet new threats, mission requirements, and coalition configurations, and
- the increasing requirements for interoperability in net-centric environments.

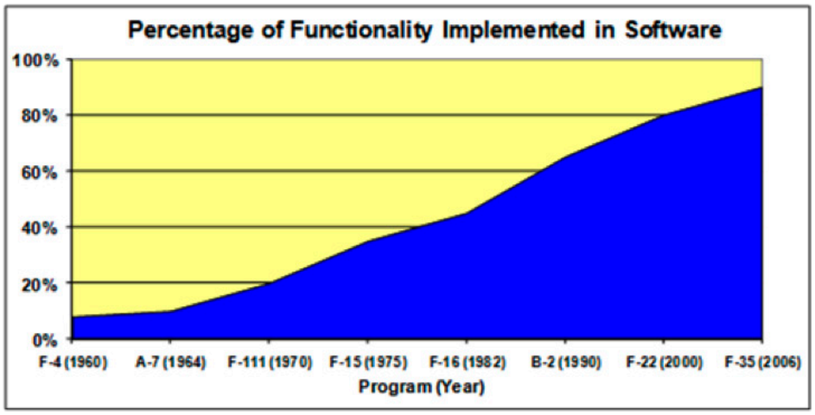


Figure 1: The Increasing Role of Software in Avionics Systems

The confluence of these and other trends impact the spectrum of acquisition and sustainment policies, programs, and infrastructure. These trends also exacerbate the growth in total ownership costs across program lifecycles [6].

Unfortunately, DoD acquisition programs have traditionally discounted design and program planning considerations for system sustainment until late in the acquisition phase (if at all). This attitude stems partly from the difficulty involved in measuring "sustainability attributes" in early phases of design and implementation. This difficulty, in turn, impedes a style of evolutionary enhancement during sustainment, where increments of investment in a system yield increments of immediate value in enhanced functionality, improved performance, etc.

Increasingly, however, the costs of software sustainment are becoming too high to discount for several reasons:

- Sustainment costs account for 60 to 90 percent of the total software lifecycle effort [5], which motivates the need to address sustainment throughout acquisition program lifecycles and improve the ability to measure—and ultimately reward—design and quality attributes applied during development that favor sustainability.
- In an era where DoD new-start programs are being reduced in favor of prolonging legacy systems, significant software sustainment cost increases are themselves unsustainable [6].

The growing expense of legacy systems—and their prolonged use—necessitate greater discipline, a sense of urgency, and attention to methods and technologies designed to improve sustainment.

To meet these challenges, software sustainment organizations must have a resilient and properly resourced infrastructure that integrates processes, practices, and people with evolving competencies, tools, information, databases, and system-integration lab capabilities. These infrastructure elements, in turn, must be systematically refreshed throughout the life of a system to support, maintain, and operate in accordance with unique properties of software in DoD systems.

For example, software does not follow the laws of physics that bound hardware design and define failure [1]. Legislative and DoD policies, however, have historically mandated a depot-centric maintenance paradigm based on relatively discrete hardware aging/replacement models. Unfortunately, these models are not well-suited to understand the cost, effort, and quality drivers of software sustainment, which is a continuing software engineering process that lasts for decades.

The Impact of Supply Chains on Software Infrastructure and Sustainment

Compared to legacy systems, newer DoD systems tend to rely more on software as a primary means to deliver functionality [1]. There are good reasons for this trend, which has rapidly accelerated over the past decade in both national security systems and commercial systems. In particular, the increasing use of—and dependency on—software means there are fewer limits on what capabilities can be enhanced and created in the future. For example, the percentage of avionics specification requirements that rely on software has risen from approximately 8 percent of the F-4 in 1960 to 45 percent of the F-16 in 1982, 80 percent of the F-22 in 2000, and 90 percent of the F-35 in 2006 [7], as shown in Figure 1.

Software is ubiquitous in DoD systems, and it is increasingly hard to identify sub-systems and components that are not controlled or enabled by software. Ironically, in this increasingly software-reliant environment, there is a growing bow wave of software sustainment demands (of unknown size, complexity, characteristics, and technical debt) that are neither recognized nor understood by the acquisition community and the DoD enterprise.

For example, not only are we dealing with a growing software base, but also the constantly evolving infrastructure in which software runs. This infrastructure includes commercial and open-source components, frameworks, and libraries, all of which are increasingly necessary for modern software systems. Moreover, there is increasing reliance on software supply chains that provide and support this infrastructure.

For example, there are supply chains for hardware/firmware components, as well as integrated components, such as network routers, operating systems, databases, and middleware [16]. A supply chain for COTS software products includes product development organizations and their suppliers. Likewise, the supply chains for custom-developed DoD acquisition systems can include the prime contractors, subcontractors, and supply chains for the COTS products used.

Software infrastructure typically evolves at a rapid pace, driven by opportunities to increase capability, improve performance, provide repairs and security enhancements, and exploit growth in underlying hardware capability. This upgraded capability must be integrated into existing systems. Likewise, software defects and performance bottlenecks must continually be identified, fixed, and optimized to provide full functionality.

Infrastructure also evolves due to improvements in its own underlying infrastructure, (i.e., lower layers of the software/hardware stack). A common example involves improvements in underlying operating systems, cloud architectures, and storage and processing capabilities that enable improvements to a database framework. An important consequence of this—and a principal driver of component-based and service-oriented software paradigms—is the speed and efficiency with which new capabilities can be manifested. For example, talented undergraduate students can apply modern software and hardware infrastructure in a matter of weeks to create highly capable mobile software apps that access dedicated cloud resources and can be widely deployed and supported.

The increasing reliance of DoD systems on software supply chains extends well beyond the defense industrial base. This trend is the subject of a 2007 Defense Science Board report [8] regarding the challenges of testing and evaluating these supply chains. Although software does not wear out, firmware and hardware become obsolete rapidly, thereby driving changes in software applications and infrastructure.

In mainstream commercial systems, these changes are planned for and provide end users a steady flow of improvements in performance and reliability derived from the underlying infrastructure. Just as importantly, these changes create headroom for improvements in function and capability.

The Relationship of Software Sustainment to Modernization Efforts

The majority of software sustainment activity is better described and managed as a modernization effort. This shift in perspective is consistent with commercial development practices and shifts in the business environment for defense systems [10]. The technical drivers discussed below—along with the ongoing rapid growth in capability of software infrastructure discussed above—have also enabled this move toward modernization.

In general, software sustainment involves the following pattern of repair, enhancement, and adaptation:

- Repair in response to defects and vulnerabilities related to functional, quality, and security attributes.
- Enhancement in response to demands for increased functional capability and performance, driven by competitive pressure (in the commercial world) and changes in mission profile (in defense).
- Adaptation in response to improvements, changes, new opportunities in the underlying stack of software and hardware infrastructure, and the mission benefits of increased interoperability among software-reliant systems in the enterprise.

This pattern is pervasive in commercial software. In recent years, this software sustainment pattern—and the tempo at which that pattern has been applied—has been amplified because many applications and data repositories have migrated to cloud-based systems [9]. This transformation is evident across the spectrum, from mobile apps (which tend to rely on cloud-based resources) to large-scale data-intensive applications.

The sustainment community has shifted from primarily emphasizing repair to focusing on enhancement and adaptation [6]. This shift stems from various mission and business considerations, not the least of which is the reduced deployment of new systems in favor of sustaining legacy systems. It is also a result of the DoD's growing ability to manifest increasing levels of functionality in software, which in turn is a consequence of the rapid pace of innovation in tools, languages, models, and processes.

Indeed, cloud-based software applications may have a much greater tempo in their update cycle. The term "DevOps" arose in the context of commercial systems to refer to the rapid iteration of development, quality assurance, and operations. This iteration is most evident in cloud-based applications due to the relative ease—and transparency—of deployment, especially when quality practices are integrated into development efforts.

Understanding and Mitigating the Cost Drivers of Software Sustainment

To craft a more effective and efficient approach to software sustainment, organizations must examine and understand the complexities and costs of the software infrastructure environment. This complex nexus of activities has historically been neglected. Recent studies [2][6], however, indicate that the DoD is expending more time and effort sustaining software, often more than originally anticipated due to uncertainties encountered during initial program cost estimation.

For example, a 2011 Air Force Scientific Advisory Board study [6] showed that total weapon system software sustainment costs have doubled in less than 10 years, as shown in Figure 2. Likewise, software sustainment hours at the three Air Logistics Centers over the past eight years have also increased significantly.

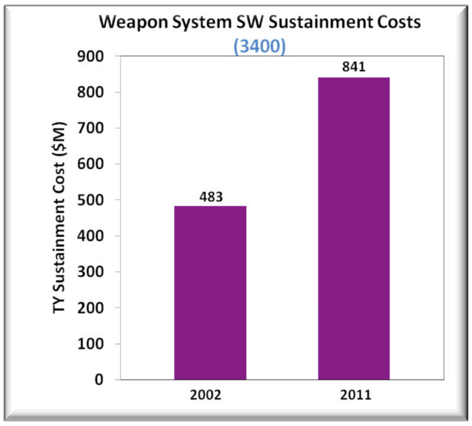


Figure 2: Increase in Software Sustainment Costs Over the Past Decade

On a larger scale, the Office of the Deputy Assistant Secretary of the Army for Cost and Economics—in collaboration with the Air Force and the Navy—is sponsoring critical and foundational research into understanding the myriad of activities that occur in what the DoD calls "software depot maintenance." SEI at Carnegie Mellon University has also initiated research [14] that addresses the uncertainty of cost estimates early in the lifecycle and the dynamics of decision making associated with choices about sustainment strategies.

Various factors contribute to the high costs of software sustainment. For example, functionality (such as fly-by-wire) originally provided by hardware may be replaced by software, which must then be sustained. Periodic software upgrades and enhancements throughout the lifecycle of DoD systems may also result in unanticipated increases in sustainment costs. Moreover, software maintainers must expend costly and time-consuming effort to understand original designs and carefully make changes to avoid degrading design integrity or negatively impacting key quality attributes. In addition, the scale and complexity of software are growing significantly to meet the expanded threat spectrum [11], which exacerbates sustainment costs.

As sustainment costs have increased, the DoD has struggled to support all its legacy systems—especially its weapon systems platforms—many of which will remain in the operational inventory much longer that planned due to budget constraints. Examples of weapon systems platforms include the physical airframes, hulls, chassis, and their associated parts such as engines, weapons, sensors, and computing/communication units. Economic strategies for understanding and addressing these rising costs are affected by a key difference between the software running in DoD weapon system platforms and the platforms themselves.

Sustainment costs have historically been attributed to the following factors:

- **the number of systems in the operational inventory,**
- **the operating tempo (optempo) of systems (flying hours, driving miles, number of deployments, etc.),**
- **the number of different configurations,**
- **parts count, and**
- **failure rates.**

The wear and tear on hardware at the platform-, sub-system-, and component-levels represents a significant maintenance expense. Through the years, the DoD has developed a finely tuned set of heuristics for estimating field and depot maintenance costs, budgets, and the relationships of maintenance funding and backlogs to operational readiness. In the face of declining budgets, the DoD has traditionally handled these costs by shrinking its force structure inventory and the operational tempo of forces, (e.g., by retiring and/or reducing the numbers of aging aircraft, ship, and vehicle platforms) [6].

This approach worked when sustainment costs were largely a function of the hardware for weapon system platforms. In contrast, software has essentially no expenses related to manufacturing or wear-and-tear. As a result, software sustainment costs are insensitive to the traditional hardware-maintenance cost drivers. In fact, software sustainment costs are primarily driven by the function a system or sub-system exists to perform, the multiple configurations of systems in the inventory (each with their own software variant), and the increasing degree of interoperability among systems in net-centric environments.

For example, a class of ships, planes, or vehicles may have scores of software variants reflecting different sensor, processing hardware, operating system, and network/bus configurations; different algorithms; and different security profiles for customers from different countries. Sustaining all these variants affects the time and effort required to assure, optimize, and manage the system throughout the lifecycle. These factors then inform the size, configuration, and capabilities required of the software sustainment infrastructure.

A critical workforce challenge is the need to reconsider current legislative and policy mandates concerning the organic and contractor share of sustainment across the DoD enterprise [6]. The pace of technological change—coupled with the continuous need to deliver greater performance to the warfighter at an affordable level of investment—creates significant pressure to assess, at the DoD-enterprise level, how to plan, organize, and perform software sustainment. This assessment should create more effective, efficient, and continually refreshed software-sustainment strategies and organizations, and the alignment of those organizations around portfolio and product-lines.

The Importance of Architecture in Enabling Effective and Efficient Sustainment

Software variability inevitably grows in legacy systems unless a concerted effort is made to rein it in. Unchecked, it becomes increasingly hard to avoid adding unnecessary variability, re-implementing variation mechanisms more than once, selecting incompatible or awkward variation mechanisms, and missing required variations. This bloat can be overcome through explicit attention to architectural features and encapsulation of the various separate dimensions of variability [12], which is a principal feature of software architecture [13].

In modern software-reliant systems, the concept of architecture includes commitments regarding the structure and content of the interactions among system components [1]. Structural commitments generally focus on which components can interact and how information exchanged between components is represented, scaled, and transmitted via data models and protocols. Other commitments may include critical quality attributes, such as performance and availability expectations, security considerations, usability, and so on.

In short, architecture is the set of critical design commitments that regulate what may and may not happen within an overall system [12]. There are two key perspectives on architecture that are essential for effective and efficient software sustainment:

- From a management perspective, architecture embodies anticipation of change: in the rapidly evolving technology infrastructure, in capabilities that will be delivered to users over a period of 5 to ten years, and in policy and business rules. Interoperability problems are evidence of missing or inadequate architectural planning, often compounded by misaligned incentives among development teams or contractors.
- From a technical perspective, architecture provides a framework for coordinating data exchange within an enterprise and for systematically addressing quality attributes. Good architectural designs anticipate change by encapsulating variability to reduce cost and risk. In this approach, change-prone areas (such as hardware and communication infrastructures) are accessed via stable interfaces whose implementations can be replaced without undue side-effects on other software components. Many software patterns [13][15] exist entirely for this purpose.

Architectural decisions thus regulate the overall interplay among systems within an enterprise. In many enterprises, "architecture" may be the result of incremental decisions over time, where a sequence of local decisions determines overall organizational outcomes, for better or worse.

Failure to attend to architecture often leads to the loss of intellectual and configuration control that is manifested via terms such as "software rot" or "bloatware." In the absence of an architecture-centered approach, the DoD will face "sticker shock" because software sustainment costs are unlikely to decrease by shrinking inventory alone. For example, since the cost drivers for software sustainment relate more to the (combinatorial) number of configurations and variants, approximately the same level of effort is needed regardless of whether there are 100 or 10,000 hardware platforms.

To address these issues, the DoD needs different strategies for understanding and alleviating rising software sustainment costs by considering architecture-based approaches early in the system-acquisition process. Architecture must therefore be an explicit consideration in the systems engineering trade-off process in advanced development planning and the technology development phase of the acquisition process. In particular, sustainment strategies based on managing software commonality and variability via software product lines should be considered when conducting systems engineering trade-off analyses [12].

Workforce Challenges Associated with Software Sustainment

In addition to the technical and economic challenges discussed above, the DoD faces challenges with recruiting, training, and retaining an efficient, productive, and continually refreshed workforce of engineers and technical managers to meet its sustainment needs [1][6]. Effective software sustainment requires this workforce to have expertise in older programming languages, operating platforms, and tools. It must also have deep domain knowledge, software architecture knowledge, and a full appreciation of the emerging software technologies that will form the basis of reengineered systems. More experienced members of the DoD workforce tend to possess this expertise, so retaining and replenishing this critical human resource is essential.

In general, the DoD's software sustainment activities rely on a combination of in-house expertise (so-called "organic sustainment") and external capability (accessed through contracting, consultancy, or advisory panels). A base of capable in-house expertise is essential in any technology-intensive organization, even those that outsource the bulk of actual technical work. In-house experts help ensure an organization is a smart customer on development projects. For example, these experts can identify needs and opportunities, create and manage relationships, structure incentives, evaluate risks and costs, and otherwise assure that the external (and internal) relationships are technically sound and aligned with organizational interests.

In-house expertise is particularly essential for DoD programs, program offices, and services to address architectural sustainment issues that transcend individual systems, development activities, and acquisition programs. These broader issues involve how separately managed, contracted development efforts might interact. While external advice can (and should) be sought and followed, it is necessary—from the standpoint of vision, strategy, and accountability—that the core technical leadership come from within the organization [1][8].

For in-house sustainment activity, a high-quality technical workforce is essential to support rapid, informed, and agile responses to evolving mission requirements, operational needs, and changes in technology infrastructure. Fewer barriers exist for in-house teams to engage in modern iterative and incremental development practices to support rapid evolution. Unfortunately, although some in-house organizations [5] are dedicated to sustaining software, their efforts are often not as well recognized (or funded) by the DoD, especially in the face of an aging DoD inventory [2].

The DoD must also address other critical deficiencies to achieve and sustain a high-quality workforce. For example, software acquisition management and software engineering are not DoD career fields, even though expertise in these domains has proven critical to success. There is thus an urgent need to address critical and emerging workforce challenges stemming from current legislative and policy mandates concerning the organic and contractor share of sustainment across the DoD enterprise.

The rapid pace of technological change, coupled with the ever-increasing need to deliver greater performance to the warfighter at an affordable level of investment, creates significant pressure to objectively assess at the DoD enterprise level how to plan, organize, and perform software sustainment. This assessment should seek to create more effective, efficient, and continually refreshed software sustainment strategies, organizations, and alignment of those organizations around portfolio and product-lines.

Key R&D Activities Needed to Address Software Sustainment Challenges

The software research community has devised various approaches to improve software sustainability. For example, tools for detecting software modularity violations help identify eroding design structures (referred to whimsically as "bad code smells" by software developers and managers) so they can be refactored. Likewise, intelligent automated regression testing frameworks help ensure that changes to legacy software work as required and that unchanged parts have not become less dependable.

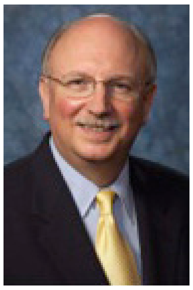
Over the past several decades, the SEI has created methods and guidelines for sustaining, migrating, and evolving legacy systems. For example, the SEI has devised strategies for modernizing legacy systems and reusing legacy components. These strategies employ risk-managed, incremental approaches that encompass changes in software technologies, engineering processes, and business practices. In addition, the SEI has created techniques for measuring the effectiveness of software-sustainment practices. These techniques can help decision-makers select between (1) continued sustainment versus replacement or (2) which of the multiple (redundant) legacy systems to keep and which to retire.

Conclusion

Despite its strategic importance to the DoD, software sustainment has received relatively little visibility and emphasis as an enterprise policy, program, and resource issue. The fact that our legacy weapon systems provide competitive advantage to the warfighter is due to the dedication and skills of the software sustainment workforce, both government and contractors, located at the services' software depot centers and at contractor facilities. We contend, however, that a greater sense of urgency is required to ensure DoD's sustainment capabilities can continue to deliver warfighter capability in the face of significant fiscal, technology, and workforce challenges [3].

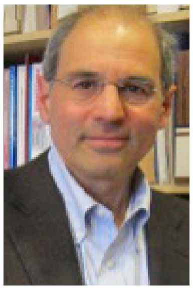
This article just scratches the surface of the complex landscape of policy, program, people, and technical design and infrastructure challenges associated with sustaining software-reliant DoD systems. Other vexing, non-technical challenges affecting sustainment and total ownership costs are that DoD contracts often fail to procure source code, necessary licenses, and technical data rights, as well as technical data on design artifacts, testing facilities, and procedures during the acquisition process [10]. The DoD needs to adopt a holistic approach to software sustainment that addresses the technical, management, and business perspectives in a balanced manner.

ABOUT THE AUTHORS



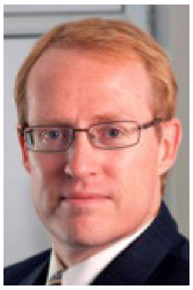
Mr. McLendon currently serves as the Associate Director, Software Solutions Division for the Software Engineering Institute, Carnegie Mellon University. Prior to assuming this position, Mr. McLendon served as Senior Advisor in the Office of the Assistant Secretary of Defense for Systems Engineering. He also served as a principal in the Office of the Assistant Secretary of Defense for Program Analysis and Evaluation and in the Office of the Under Secretary of Defense for Policy. He later was a Professor at the Defense Systems Management College. He served as a career Air Force officer in a range of leadership and management positions in system and technology development and acquisition as well as the federal level and the private sector.

SEI Carnegie Mellon University
4500 Fifth Ave
Pittsburgh, PA 15213
E-mail: mmclendon@sei.cmu.edu



Dr. William L Scherlis is a Professor in the School of Computer Science at Carnegie Mellon. He is director of CMU's Institute for Software Research (ISR) in the School of Computer Science and the founding director of CMU's PhD Program in Software Engineering. From Jan 2012 to January 2013 he served as the Acting CTO for the Software Engineering Institute. His research relates to software assurance, software analysis, and assured safe concurrency. Dr. Scherlis chaired the National Research Council (NRC) study committee on defense software producibility, which released its final report Critical Code: Software Producibility for Defense in 2010. He is a Fellow of the IEEE and a lifetime National Associate of the National Academy of Sciences. Dr. Scherlis joined the Carnegie Mellon faculty after completing a Ph.D. in Computer Science at Stanford University, a year at the University of Edinburgh (Scotland) as a John Knox Fellow, and an A.B. at Harvard University.

SEI Carnegie Mellon University
4500 Fifth Ave
Pittsburgh, PA 15213
E-mail: scherlis@sei.cmu.edu



Dr. Douglas C. Schmidt is a Professor of Computer Science and Associate Chair of the Computer Science and Engineering program at Vanderbilt University. He is also a Visiting Scientist at the Software Engineering Institute, where he served as the CTO from September 2010 to December 2011. Dr. Schmidt has published 10 books and more than 500 technical papers on software-related topics, including patterns, optimization techniques, and empirical analyses of object-oriented frameworks and domain-specific modeling environments that facilitate the development of distributed real-time and embedded middleware and mission-critical applications running over data networks and embedded system interconnects. Dr. Schmidt received B.S. and M.A. degrees in Sociology from the College of William and Mary in Williamsburg, Virginia, and an M.S. and a Ph.D. in Computer Science from the University of California, Irvine (UCI).

SEI Carnegie Mellon University
4500 Fifth Ave
Pittsburgh, PA 15213
E-mail: dschmidt@sei.cmu.edu

REFERENCES

1. National Research Council's Critical Code: Software Producibility for Defense report, <http://www.nap.edu/openbook.php?record_id=12979&page=R1>
2. National Research Council's, Aging of U.S. Air Force Aircraft report <http://www.nap.edu/catalog.php?record_id=5917>
3. Ashton Carter, "Better Buying Power: Guidance for Obtaining Greater Efficiency and Productivity in Defense Spending," Memorandum for Acquisition Professionals, September 14, 2010.
4. Mary Ann Lapham, "Sustaining Software-Intensive Systems," SEI technical report, <http://www.sei.cmu.edu/library/abstracts/reports/06tn007.cfm>
5. United States Air Force Software Technology Support Center, "Guidelines for Successful Acquisition and Management of Software-Intensive Systems: Weapon Systems, Command and Control Systems, and Management Information Systems (Condensed Version 4.0).," Ogden Air Logistics Center Hill AFB, UT, February 2003.
6. Air Force Science Advisory Board's Sustaining Aging Aircraft report, <http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA562696>.
7. Report from the Defense Science Board Task Force on Defense Software, November 2000.
8. Report of the Defense Science Board Task Force on Mission impact of Foreign Influence on DoD Software, September 2007.
9. Teri Takai, et al., "Department of Defense Cloud Computing Strategy," July 12, 2012.
10. Nick Guertin and Brian Womble, "Competition and the DoD Marketplace," Proceedings of the Ninth Annual Acquisition Research Symposium, April 30th, 2012.
11. Lind Northrop, et al., Ultra-Large-Scale Systems: The Software Challenge of the Future, Software Engineering Institute, 2006.
12. Paul Clements and Linda Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.
13. Frank Buschmann, et al., Pattern-Oriented Software Architecture: A System of Patterns, Wiley, 1996.
14. Robert Ferguson, "An Investment Model for Software Sustainment," SEI Blog, July 22, 2013.
15. Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
16. Robert J. Ellison, Christopher Alberts, Rita Creel, Audrey Dorofee, and Carol Woody, "Software Supply Chain Risk Management: From Products to Systems of Systems," CMU/SEI-2010-TN-026.

Software Sustainment Now and Future

Mary Ann Lapham, SEI

Abstract. Today's systems are increasingly reliant on software which must be sustained into the future. To sustain these systems organizations must define sustainment, meet criteria to enter sustainment, and overcome some classic sustainment challenges. This article discusses these tasks along with historical parallel development and sustainment and potential future trends in software sustainment.

Introduction

This article provides an overview of current software sustainment practices and challenges within the Department of Defense (DoD) and a look at the potential future of software sustainment within the federal government. It takes a broad view based on a specific study done in 2006 which was not meant to be all inclusive for every software sustainment topic. Thus there are areas not covered that may be relevant to an individual situation. Areas such as open source software, anti-tamper, sustainment cost estimation, and specific authority and responsibility for transition to sustainment should be explored if relevant to your situation.

As today's systems become increasingly reliant on software, the issues surrounding sustainment become increasingly complex. The risks of ignoring these issues can potentially undermine the stability, enhancement, and longevity of systems in the field.

At the center of this puzzle are disparate definitions. Developers and acquirers have a general understanding that sustainment involves modifying systems and deploying changes to meet customer needs, but does this understanding align with common practice and the DoD's definition of sustainment? DoD Instruction 5000.02 describes sustainment as follows:

Life-cycle sustainment considerations include supply; maintenance; transportation; sustaining engineering; data management; configuration management; Human Systems Integration (HSI); environment, safety (including explosives safety), and occupational health; protection of critical program information and anti-tamper provisions; supportability; and interoperability [1, section 8.c.1.b].

The terms software maintenance and software sustainment are often used interchangeably. It is important to make sure that all stakeholders use the same terminology when discussing software sustainment.

The IEEE Standard Glossary of Software Engineering Terminology defines "software maintenance" as follows:

The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment [2].

Software maintenance consists of correcting faults, improving performance or other attributes, and adapting to a changing organization and technical environment. To be complete, there is usually a fourth category of maintenance activities focused on anticipated problems, or preventive maintenance[1] [3].

While DoD Instruction 5000.02 describes sustainment in detail, no authoritative definition of "software sustainment" exists. The SEI's working definition is as follows:

The processes, procedures, people, material, and information required to support, maintain, and operate the software aspects of a system.

Given this definition, software sustainment addresses other issues not always an integral part of maintenance such as documentation, operations, deployment, security, configuration management, training (users and sustainment personnel), help desk, commercial off-the shelf (COTS) product and license management, and technology refresh. Successful software sustainment consists of more than modifying and updating source code. It also depends on the experience of the sustainment organization, the skills of the sustainment team, the adaptability of the customer, and the operational domain of the team. Thus, software maintenance as well as operations should be considered part of software sustainment.

Criteria to Enter Sustainment

The Operations and Support phase of the Defense Acquisition Management System has two major efforts, Life-Cycle Sustainment and Disposal. The entrance criteria include an approved Capabilities Production Document (CPD), an approved Life-Cycle Sustainment Plan (LCSP), and a successful Full-Rate Production (FRP) decision [1, section 8.a and b]. In addition, the following criteria among others should be considered:

• Stable software production baseline—Most sustainment organizations will not accept software into sustainment until the software is stable. Merriam-Webster Online defines stable as "a. firmly established: fixed, steadfast; b. not changing or fluctuating: unvarying; c. permanent, enduring" [4]. However, in the realm of software stable can mean different things.

If one were to apply Merriam-Webster's definition to software, he or she could infer that a single instance of loss of availability or a system failure would indicate that the software is not stable. In other words, software is stable only if it does not have problems that cause it to stop working. For software, unfortunately, the definition of stable can be a subjective one from several different perspectives. One organization may be willing to accept software as stable if it only fails once a week, while others would deem this rate of failure too high and would not accept the software. In other situations, software may be considered stable if no Category 1 or 2 Software Trouble Reports (STRs) exist.

Defining the stability of a system depends somewhat on its intended use, its mission criticality, and the potential consequences if the system fails. For instance, a system such as navigation software or command and control software whose failure could result in loss of life should have more stringent requirements for maintaining stability than one that is business software supporting actions that could be postponed for hours or even days.

The program office should define the criteria for accepting a system as stable in the Sustainment Transition Plan. These criteria should at the very least identify the types of STRs allowed to be active in a system that is entering sustainment.

• Complete and current software documentation—Complete, current software documentation is paramount for the software sustainment organization. Without it, the sustainment organization has limited insight into how the software was designed and implemented. Incompleteness or omissions increase software maintenance costs because software engineers have to reverse-engineer the code to determine how it works. In addition, this process increases the risk of inadvertently introducing errors into the code. Well documented code is a plus and—for those using incremental and iterative methods—expected.

The program office should determine what constitutes complete documentation for its system. At a minimum the documentation set should include the "why, how, what, and where" of the system as built. That is, documents should allow the sustainment organization to understand why the system was designed, how the system was developed, what the system consists of, and where functions were allocated to different subsystems. The overall architecture or blueprint for the system needs to be provided. Plans on how the program office intended to handle COTS and configuration management issues are essential for sustainment and continued implementation. Interface definitions need to be documented. Database designs and their documentation are essential to understanding their purpose within the system. Lastly, the development environment needs to be defined so the sustainment organization knows what tools were used to develop and support the system.

• Authority to Operate (ATO) for an operational software system—Before a system can be considered operational in the field and thus meet the criteria to enter sustainment, an Authority to Operate must be issued. The ATO issuance depends on approval of security requirements by the Designated Approval Authority (DAA). Issuing an ATO means that a DAA has accepted that operation of the system represents a low security risk. An ATO is issued for a fixed period of time (typically three years) and must be renewed. Delay in obtaining ATO approval or renewal could cause the system to be deemed non-operational.

• Current and negotiated Sustainment Transition Plan—In many instances, a program has been developed, tested, and declared operational but there is no funding set aside to address creation and subsequent negotiation of the Sustainment Transition Plan. Unfortunately, in an era where budgets are becoming increasingly tight, sustainment planning is postponed and in some instances forgotten.

Both the development organization and the sustainment organization need to be involved in creating the Sustainment Transition Plan. If a contractor is involved in development, that organization also needs to participate in the development and subsequent negotiation of the Sustainment Transition Plan. In addition, the contract should include tasks that address the contractor's role in the sustainment planning and transition process.

The program office should ensure that while the program is being developed, sustainment tasks are not forgotten or removed from the development contractor's tasking. While the development contractor may not necessarily become the sustainment organization, the development contractor is responsible for developing and maintaining documentation that the sustainment organization will need. It is the program office's responsibility to ensure that the contractor does not create documentation that is proprietary or undeliverable. Even though it was cancelled in 1998, the MIL-STD-498, Section 5.13, "Preparing for Software Transition," contains good background and reference material in this area [5].

• Sustainment staffing and training plan—Staffing the sustainment organization is critical. The staff needs to be trained software professionals that can work with the development organization to transfer the necessary system knowledge. One should not assume that any of the development organization staff will transition to the sustainment organization; rather, adopt a plan to transfer the knowledge from one organization to the other as part of the staffing plan. The staffing and training plan are related to and should be coordinated with the Sustainment Transition Plan.

As with many other areas associated with sustainment, training for the sustainment organization is often treated as an afterthought and is usually an under-funded activity. Even though the sustainment staff is composed of trained software professionals, they still need training on the specifics of the system entering sustainment. This is especially true for the increasingly complex systems that contain a mixture of COTS, government off-the-shelf (GOTS), and organic (government-developed) software code. "On-the-job" training is not sufficient for personnel sustaining these types of complex systems. The system's specific architecture, design decisions, and other nuances need to be communicated in some depth.

Sustainment Challenges

Our research in the 2006 timeframe identified a variety of issues or challenges prevalent with software sustainment at that time. These were grouped into six categories. This is not to say that one would not find other issues that must be addressed when a system is entering sustainment. In addition, no priority is implied by the order in which these topics are discussed.

The following categories of sustainment challenges were identified:

• Sustainment with COTS software—requires consideration of system obsolescence, technology refresh, source code escrow, and vendor license management among related topics.

• Programmatic considerations—discusses issues with relegating the sustainment requirement to the category of "minor requirements."

• System transition to sustainment—considers topics of support database transition, development and software support environment infrastructure (software test lab, hardware spares, release processes and procedures), staffing, operations training, and transition planning

• User support—discusses help desk, user documentation, and user training.

• Information assurance—discusses the unique challenges of IA and COTS software products and testing for IA.

• Development versus sustainment.

While these challenges are most likely still valid, the only one discussed in depth within this article is the last one, development versus sustainment.[2]

Parallel Development and Sustainment—History

As I found in 2006 (and continuing to the present), many systems are fielded in an incremental manner. Incremental means that an increment or version of the system that provides partial capabilities is developed and fielded. The remaining capabilities are developed later depending on budget, requirements definition, and technology advancements. For the sustainment organization, this means that it will be sustaining a system in parallel with another version of the system that is still under development.

Development in parallel with sustainment is not a new concept; however, many sustainment organizations may not have experience with this mode of operation. In some instances, upgrades (development) are considered a sustainment activity. This makes the "line" between development and sustainment very hazy. To ensure continued operation of the system, the sustainment and development organizations need to develop processes and procedures, coordinate them with all parties, and obtain concurrence on their use. This should include an understanding of who is responsible for any upgrades.

Historically, in organizations that are successful in performing development and sustainment concurrently, groups within the organizations report to the same person. Given the organizational structure of the development and sustainment organizations, this can be problematic. In many instances, the person who has enough experience to oversee both the development and sustainment groups does not have the desire or the time to be involved in this level of oversight.

To better align parallel development and sustainment efforts, the program office needs to consider the current sustainment structure. With that in mind, it should then determine how the system being developed is evolving and how it can fit into the sustainment structure. Sustainment organizations should plan to adapt their processes to handle an evolving system, especially if it implements COTS hardware or software products.

In addition, a joint (development and sustainment) Configuration Control Board (CCB) needs to be created and given the authority to act. All decisions for changes to the baseline must go through the CCB **without exception**. The operational software **must** be driven from the CCB approved baseline. Last, a clear, documented path of escalation up to senior-level personnel **must** be created to address issues. It is not a question of if there will be issues, but when they will occur. Being prepared to handle issues reduces the impact problems have on the overall development and sustainment of the system. Emphasis in bold is added to point out the criticality of following a strict CCB process when there are two organizations (one development focused and one sustainment focused). Otherwise, keeping the two systems in alignment will be problematic at best.

Future of Software Sustainment

In 2006, when I authored the Sustaining Software Intensive Systems technical note, many commercial organizations were starting to use incremental and iterative methods known as Agile methods. These methods have evolved over time; today the commercial environment is using something referred to as DevOps. What is DevOps?

What it is. A way of working that encourages the Development and Operations teams to work together in a highly collaborative way towards the same goal.

What it is not. A way to get developers to take on operational tasks and vice versa [6].

Strangely, I find the definition very similar to what was described in the Parallel Development and Sustainment section. However, there are some major differences. DevOps seems to be the Agile community's term for doing sustainment and operations in parallel. The methods used are based on the Agile Manifesto four tenets and 12 principles but applied in a sustainment environment. Adopting these tenets and principles within DoD requires a major change in the paradigm for doing business [7].

The SEI currently has a team researching the use of Agile methods in sustainment within the federal government. This research is how I came upon the term DevOps. In addition, Gene Kim provided a keynote speech on DevOps at the 2013 Software Technology Conference. The question is whether this type of methodology will be useful and adopted within the federal government. We're still trying to determine this.[3]

However, we have learned that several maintenance organizations within the federal government are trending toward using more Agile-like methods for conducting sustainment. While the "jury is still out" on whether Agile methods are indeed in use, there seems to be a movement to try more incremental and iterative methods using empowered teams. This movement toward incremental and iterative methods does seem to make sense for a sustainment environment where defects and/or enhancements are prioritized and worked on in that order based on the amount of capacity the sustainment team possesses. This approach sounds eerily like the product backlog maintained by an Agile team [8].

In fact, one of the early conclusions by SEI in our Agile work included the following thoughts on using Agile for sustainment:

Operations and Support is where sustainment of the software is conducted. It is assumed that the software previously developed (during the Engineering and Manufacturing Development phase) is mature and stable, so the anticipated software effort expended during this phase is low and should follow a sustainment model, driven by the need to correct errors observed during qualification testing, or providing enhancements as requested by program stakeholders. It is quite possible for a software development team working in these life cycle phases to follow an Agile approach. Quite often the features requested during this phase are modifications that are only relevant within the context of the system that had been previously developed. The aspect of user involvement that naturally occurs at this point of the life cycle makes it easier for the use of a collaborative approach.

It should be noted that some of the Agile methods might not be as practical as others[4] during the Operations and Support phase. For example, it is quite likely that the capability provided during sustainment is planned to be provided over a significant period of time, typically on the order of two years. While the involvement of the user might be beneficial, the frequent releases may not be useful because of limitations with the verification and validation environments required for deployed systems. On the other hand, this constraint should not preclude the use of Agile during this stage of development [8].

In addition, many issues need to be explored including but not limited to documentation required, CCB interaction, release of updated software to the field, quality of code, and cost of code.

Our ongoing Agile and sustainment research is looking at these and other issues. The results of our Agile and sustainment study should be available in early 2014.

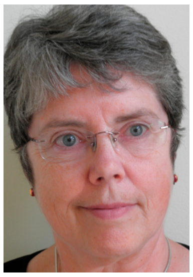
Summary

There are multiple issues associated with software sustainment. They start with agreeing on a standard definition for the term software sustainment. This is followed by knowing the criteria for entering sustainment which include stable software production baseline; complete and current software documentation; Authority to Operate; current and negotiated Sustainment Transition Plan; and sustainment staffing and training plan. Finally, specific known challenges need to be considered. These include but are not limited to sustainment with COTS software; programmatic considerations; system transition to sustainment; user support; information assurance; and development versus sustainment.

Parallel development and sustainment have historically been done which may lead to a move towards the more current DevOps approach. DevOps is becoming popularized by the Agile movement. Many issues need to be resolved and the jury is still out on the effectiveness of this approach in the federal government.

Disclaimers:

ABOUT THE AUTHOR



Mary Ann Lapham, a Principal Engineer at the Software Engineering Institute (SEI) of Carnegie Mellon University, is the technical lead for SEI's agile in acquisition research, focused on identifying and addressing barriers to adopting Agile practices in DoD and other government settings. She is also the Space Sector lead within the Software Solutions Division, Client Technical Solutions Directorate. Prior to her coming to the SEI in 2004, Ms. Lapham spent 30 years in technical and program management roles on programs of variable size and complexity. She also is a PMP and CSM.

E-mail: mlapham@sei.cmu.edu
Phone: 412-268-5498

REFERENCES

1. Department of Defense. DoD Instruction Operation of the Defense Acquisition System (DoDI 5000.02). December 2008. Print.
2. Institute of Electrical and Electronics Engineers. IEEE Standard Glossary of Software Engineering Terminology (IEEE Std. 610.12-1990). New York, NY: IEEE, 1990 (ISBN: 155937067X). Print.
3. Lapham, M.A.; Woody, C. Sustaining Software Intensive Systems (CMU/SEI-2006-TN-007). Software Engineering Institute, Carnegie Mellon University. Web. 2006. <http://www.sei.cmu.edu/library/abstracts/reports/06tn007.cfm>
4. "Stable." Merriam-Webster Online Dictionary, 10th Edition. Web. <http://www.merriam-webster.com/dictionary/stable>
5. Department of Defense. MIL-STD-498 Software Development and Documentation. December 1994. (Cancelled June 1998). Print.
6. Swartout, Paul. Continuous Delivery and DevOps: A Quickstart Guide, Continuous Delivery and DevOps Explained. Packt Publishing, 2012. Print.
7. Lapham, M.A.; Miller, S; Adams, L; Brown, N; Hackemack, B; Hammons, C; Levine, L; and Schenker, A. Agile Methods: Selected DoD Management and Acquisition Concerns (CMU/SEI-2011-TN-002). Software Engineering Institute, Carnegie Mellon University. Web. 2011. <http://www.sei.cmu.edu/library/abstracts/reports/11tn002.cfm>
8. Lapham, M.A.; Williams, R.; Hammons, C.; Burton, D.; & Schenker, A. Considerations for Using Agile in DoD Acquisition (CMU/SEI-2010-TN-002). Software Engineering Institute, Carnegie Mellon University. Web. 2010. <http://www.sei.cmu.edu/library/abstracts/reports/10tn002.cfm>

NOTES

1. Information for sustainment is based on the SEI Technical Note Sustaining Software-Intensive Systems, CMU/SEI-2006-TN-007 and updated to reflect the DoDI 5000.02 released in 2008
2. For discussion on the first five challenges see Sustaining Software-Intensive Systems, CMU/SEI-2006-TN-007, http://www.sei.cmu.edu/library/abstracts/reports/06tn007.cfm
3. A future technical note is expected to be published in early 2014 addressing agile and sustainment topics.
4. Kanban/lean style of Agile might be the most relevant for this phase.

Upcoming Events

Visit <http://www.crosstalkonline.org/events> for an up-to-date list of events.

International Conference on Computing, Networking and Communication
2-6 February 2014
Honolulu, HI
http://www.wikicfp.com/cfp/servlet/event.showcfp?eventid=30749&copyownerid=548

2014 T3 Advisor Conference
10-12 February 2014
Anaheim, CA
http://2014t3-eorg.eventbrite.com

Spring 2014 Software & Supply Chain Assurance Forum
11-13 March 2014
McLean, VA
https://buildsecurityin.us-cert.gov/swa

30th Annual National Test & Evaluation Conference
24-27 March 2014
Seattle, WA
http://www.ndia.org/meetings/4910/Pages/default.aspx

Summer 2014 Software & Supply Chain Assurance (SSCA)
Working Group Sessions
24-26 June 2014
McLean, VA
https://buildsecurityin.us-cert.gov/swa

American Society of Quality International Conference
February 25-26, 2014
Dallas, TX
http://asq-icsq.org

SUBSCRIBE TODAY!

To subscribe to CrossTalk, visit www.crosstalkonline.org and click on the subscribe button.

Form. Fit. Function.

I found myself with a looming deadline (basically, this column was due five days ago) and was scrambling to come up with a good topic that fits in with “Legacy Systems Software.” I kept delaying, and decided that I would write the column when I flew to New York for a meeting. I was waiting for inspiration. As luck would have it, as I was texting my wife from the airplane, the flight attendant gently reminded me that it was time to turn off all electronic equipment.

And there it was. You see, I am a relatively proud user of an…… well, to keep from getting sued, let us say I have a “MiPhone.” In fact, I have been a MiPhone user since 2009. I also have a MiPad, and several MiMacs (two laptops and a desktop at the office). I have been a loyal MiMac user since 1988, and prefer it to the alternative operating system.

I teach computer science, and 30% of my students use Macs. I feel I have an obligation to show students how the two major desktop operating systems compare. On a PC, I cannot install Mac OS. However, on all three of my Macs, I can run several virtualized operating systems simultaneously. I can show how to accomplish some task on Mac OS, and then quickly show the same task on Windows 7, and then on Windows 8 with just a simple swipe to another virtual environment. To me, a computer is simply a tool, and right now, with the job I have, a Mac is the right tool for me.

Back to my MiPhone. Since I bought it back in 2009, it has basically operated the same. I got it when MiOS 3 was out, and through MiOS 4, 5, and 6, it has basically operated the same. Each successive operating system brought out new features, but the older features basically worked exactly the same. So much so, that my brain trained itself to run on autopilot. Need to unlock the phone? My thumb knew where to push to enter my massively secure 4-digit password. Need to go to mail? Once loaded, my thumb automatically knew how to read an email, and then swipe to delete it.

Need to set an alarm? The MiPhone alarm had two dials to set the hour and minute, and my thumb, over the last five years, automatically knew how hard to swipe it to get it to roll from 15 minutes after the hour all the way around to 45 after. In fact, like so many others, my phone was so much a part of my life that I automatically grabbed for it, unlocked it, and clicked an icon without thinking—until last week.

I am writing this column the last week of September, at which time, the long-awaited MiOS 7 was released. And while adding lots of cool features, it also changed a lot of existing features.

The unlock screen changed both the size and location of the number pad. Granted, it only takes a while to figure out the new positioning, but why is there a new size and layout of the number pad?

For some reason, the icons for certain long-used applications (such as Photos) have totally changed. In fact, there is a general redesign of almost all of the graphics. Everything somehow looks childish and less colorful. Again, I have to ask—why? For almost five years, I automatically knew what the icon for Settings looked like. It was like being on autopilot to find it. How hard is it to have to re-learn what the Settings (and Photos, and several other apps) icons look like? But the bottom line is, why should I have to re-learn what I already knew well?

One last complaint. To get to the Spotlight search screen, you used to go to the home screen and swipe left to right. MiPhone users learned to quickly hit the select button twice (bringing you to the first home screen) and then swiping. Now, however, you swipe down from the middle of any screen to get to search. I agree, the new method is better. But they disabled the older method. The older way could have been left working. Now I have to retrain my muscle memory. For five years I have automatically clicked twice and swiped without thinking about it.

Form. Fit. Function. These are the keys to sustainable legacy software. Legacy software must evolve, but have the same basic form. The fit must match existing interfaces. And any new function should not violate rules that the user has spent years and years learning. It is ok to make users learn new things, but do not make them unlearn what has always worked. Any new functionality should not delete old functionality (assuming the old functionality was not incorrect).

Imagine a Windows computer where the three keys for the task manager became Tab-Shift-Return. How long would it take you to quit hitting Ctrl-Alt-Delete by reflex?

My wife, who has owned both a MiPhone and MiPad longer than me, summed it up nicely when she said, “If I am going to have to relearn basic functionality, why not just relearn on an Android instead of a MiPhone?”

Change it too much and your users might start looking for newer alternatives. Be it an Android or large-scale software. One thing is for sure, your users certainly won’t be happy.

Form. Fit. Function. Even on an inexpensive phone.

David A. Cook, Ph.D.
Stephen F. Austin State University
cookda@sfasu.edu

HILL AIR FORCE BASE IS HIRING

SOFTWARE ENGINEERS AND COMPUTER SCIENTISTS

EXCITING AND STABLE WORKLOADS:

- ★Joint Mission Planning System
- ★Battle Control System-Fixed
- ★Satellite Technology
- ★Expeditionary Fighting Vehicle
- ★F-16, F-22, F-35, New Workloads Coming Soon
- ★Ground Theater Air Control System
- ★Human Engineering Development

EMPLOYEE BENEFITS:

- ★Health Care Packages
- ★10 Paid Holidays
- ★Paid Sick Leave
- ★Exercise Time
- ★Career Coaching
- ★Tuition Assistance
- ★Retirement Savings Plans
- ★Leadership Training

LOCATION, LOCATION, LOCATION:

- ★25 minutes from Salt Lake City
- ★Utah Jazz Basketball
- ★Three Minor League Baseball Teams
- ★One Hour from 12 Ski Resorts
- ★Minutes from Hunting, Fishing, Water Skiing, ATV Trails, Hiking

Like facebook
www.facebook.com/309SoftwareMaintenanceGroup

CONTACT US:

Email: *309SMXG.SODO@hill.af.mil*
Phone: (801) 775-5555

CROSSTALK thanks the above organizations for providing their support.